O'REILLY®

Seventh Edition

JavaScript

The Definitive Guide

Master the World's Most-Used Programming Language

David Flanagan

O'REILLY®

JavaScript: The Definitive Guide

JavaScript is the programming language of the web and is used by more software developers today than any other programming language. For nearly 25 years this best seller has been the go-to guide for JavaScript programmers, and this new edition has been fully updated to cover the 2020 version of JavaScript. You'll find engaging and illuminating example code throughout and new chapters covering classes, modules, iterators, generators, Promises, and async/await.

This book is for programmers who want to learn JavaScript and web developers ready to take their understanding and mastery of the language to the next level.

Topics include:

- Types, variables, operators, statements, objects, and arrays
- Functions, classes, modules, iterators, generators, Promises, and async/await
- JavaScript's standard library: data structures, regular expressions, JSON, internationalization, and URLs
- The web platform: documents, components, graphics, networking, storage, and threads
- Node.js: buffers, files, streams, threads, child processes, web clients, and web servers
- Tools and language extensions that professional JavaScript developers rely on

David Flanagan has been programming with and writing about JavaScript since 1995. He has a degree in computer science and engineering from the Massachusetts Institute of Technology and works as a software engineer at VMware.

"This book is everything you never knew you wanted to know about JavaScript. Take your JavaScript code quality and productivity to the next level. David's knowledge of the language, its intricacies, and its gotchas is astounding and it shines through in this truly definitive guide to the JavaScript language."

—**Schalk Neethling**
Senior Frontend Engineer
at MDN Web Docs

"David Flanagan takes readers on a guided tour of JavaScript that will provide them with a feature-complete picture of the language and its ecosystem."

—**Sarah Wachs**
Frontend Developer and
Women Who Code Berlin Lead

WEB PROGRAMMING / JAVASCRIPT

US $69.99 CAN $92.99

ISBN: 978-1-491-95202-3

9 781491 952023 56999

Twitter: @oreillymedia
facebook.com/oreilly

Praise for *JavaScript: The Definitive Guide*, Seventh Edition

"This book is everything you never knew you wanted to know about JavaScript. Take your JavaScript code quality and productivity to the next level. David's knowledge of the language, its intricacies and gotchas, is astounding, and it shines through in this truly definitive guide to the JavaScript language."

—Schalk Neethling, Senior Frontend Engineer at MDN Web Docs

"David Flanagan takes readers on a guided tour of JavaScript that will provide them with a feature-complete picture of the language and its ecosystem."

—Sarah Wachs, Frontend Developer and Women Who Code Berlin Lead

"Any developer interested in being productive in codebases developed throughout JavaScript's lifetime (including the latest and emerging features) will be well served by a deep and reflective journey through this comprehensive and definitive book."

—Brian Sletten, President of Bosatsu Consulting

SEVENTH EDITION

JavaScript: The Definitive Guide

Master the World's Most-Used Programming Language

David Flanagan

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

JavaScript: The Definitive Guide, Seventh Edition
by David Flanagan

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://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Jennifer Pollock
Development Editor: Angela Rufino
Production Editor: Deborah Baker
Copyeditor: Holly Bauer Forsyth
Proofreader: Piper Editorial, LLC

Indexer: Judith McConville
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

June 1998: Third Edition
November 2001: Fourth Edition
August 2006: Fifth Edition
May 2011: Sixth Edition
May 2020: Seventh Edition

Revision History for the Seventh Edition
2020-05-13: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491952023* for release details.

978-1-491-95202-3

[LSI]

To my parents, Donna and Matt, with love and gratitude.

Table of Contents

Preface

This book covers the JavaScript language and the JavaScript APIs implemented by web browsers and by Node. I wrote it for readers with 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. My goal with this book is to document the JavaScript language comprehensively and definitively and to provide an in-depth introduction to the most important client-side and server-side APIs available to JavaScript programs. As a result, this is a long 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.

Previous editions of this book included a comprehensive reference section. I no longer feel that it makes sense to include that material in printed form when it is so quick and easy to find up-to-date reference material online. If you need to look up anything related to core or client-side JavaScript, I recommend you visit the MDN website. And for server-side Node APIs, I recommend you go directly to the source and consult the Node.js reference documentation.

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 occasionally used when explaining JavaScript syntax.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user

This element signifies a general note.

This element indicates a warning or caution.

Example Code

Supplemental material (code examples, exercises, etc.) for this book is available for download at:

https://oreil.ly/javascript_defgd7

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us 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 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.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*JavaScript: The Definitive Guide*, Seventh Edition, by David Flanagan (O'Reilly). Copyright 2020 David Flanagan, 978-1-491-95202-3."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

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

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

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/javascript_defgd7*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and more information about our books and courses, 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 with the creation of this book. I'd like to thank my editor, Angela Rufino, for keeping me on track and for her patience with my missed deadlines. Thanks also to my technical reviewers: Brian Sletten, Elisabeth Robson, Ethan Flanagan, Maximiliano Firtman, Sarah Wachs, and Schalk Neethling. Their comments and suggestions have made this a better book.

The production team at O'Reilly has done their usual fine job: Kristen Brown managed the production process, Deborah Baker was the production editor, Rebecca Demarest drew the figures, and Judy McConville created the index.

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, Mike Loukides, Nick Thompson, Norris Boyd, Paula Ferguson, Peter-Paul Koch, Philippe Le Hegaret, Raffaele Cecco, Richard Yaker, Sanders Kleinfeld, Scott Furman, Scott Isaacs, Shon Katzenberger, Terry Allen, Todd Ditchendorf, Vidur Apparao, Waldemar Horwat, and Zachary Kessin.

Writing this seventh edition 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, March 2020

CHAPTER 1

Introduction to JavaScript

JavaScript is the programming language of the web. The overwhelming majority of websites use JavaScript, and all modern web browsers—on desktops, tablets, and phones—include JavaScript interpreters, making JavaScript the most-deployed programming language in history. Over the last decade, Node.js has enabled JavaScript programming outside of web browsers, and the dramatic success of Node means that JavaScript is now also the most-used programming language among software developers. Whether you're starting from scratch or are already using JavaScript professionally, 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, interpreted programming language that is well-suited to object-oriented and functional programming styles. JavaScript's variables are untyped. Its syntax is loosely based on Java, but the languages are otherwise unrelated. JavaScript derives its first-class functions from Scheme and its prototype-based inheritance from the little-known language 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 quite 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 suitable for serious software engineering and projects with huge codebases.

JavaScript: Names, Versions, and Modes

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." In practice, everyone just calls the language JavaScript. This book uses the name "ECMAScript" and the abbreviation "ES" to refer to the language standard and to versions of that standard.

For most of the 2010s, version 5 of the ECMAScript standard has been supported by all web browsers. This book treats ES5 as the compatibility baseline and no longer discusses earlier versions of the language. ES6 was released in 2015 and added major new features—including class and module syntax—that changed JavaScript from a scripting language into a serious, general-purpose language suitable for large-scale software engineering. Since ES6, the ECMAScript specification has moved to a yearly release cadence, and versions of the language—ES2016, ES2017, ES2018, ES2019, and ES2020—are now identified by year of release.

As JavaScript evolved, the language designers attempted to correct flaws in the early (pre-ES5) versions. In order to maintain backward compatibility, it is not possible to remove legacy features, no matter how flawed. But in ES5 and later, programs can opt in to JavaScript's *strict mode* in which a number of early language mistakes have been corrected. The mechanism for opting in is the "use strict" directive described in §5.6.3. That section also summarizes the differences between legacy JavaScript and strict JavaScript. In ES6 and later, the use of new language features often implicitly invokes strict mode. For example, if you use the ES6 `class` keyword or create an ES6 module, then all the code within the class or module is automatically strict, and the old, flawed features are not available in those contexts. This book will cover the legacy features of JavaScript but is careful to point out that they are not available in strict mode.

To be useful, every language must have a platform, or standard library, for performing things like basic input and output. The core JavaScript language defines a minimal API for working with numbers, text, arrays, sets, maps, and so on, 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.

The original host environment for JavaScript was a web browser, and this is still the most common execution environment for JavaScript code. The web browser environment allows JavaScript code to obtain input from the user's mouse and keyboard and

by making HTTP requests. And it allows JavaScript code to display output to the user with HTML and CSS.

Since 2010, another host environment has been available for JavaScript code. Instead of constraining JavaScript to work with the APIs provided by a web browser, Node gives JavaScript access to the entire operating system, allowing JavaScript programs to read and write files, send and receive data over the network, and make and serve HTTP requests. Node is a popular choice for implementing web servers and also a convenient tool for writing simple utility scripts as an alternative to shell scripts.

Most of this book is focused on the JavaScript language itself. Chapter 11 documents the JavaScript standard library, Chapter 15 introduces the web browser host environment, and Chapter 16 introduces the Node host environment.

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 related material. This introductory chapter makes a quick first pass through the language, introducing key features that will make it easier to understand the in-depth treatment in the chapters that follow. If you are already a practicing JavaScript programmer, you can probably skip this chapter. (Although you might enjoy reading Example 1-1 at the end of the chapter before you move on.)

1.1 Exploring JavaScript

When learning a new programming language, it's important to try the examples in the book, then modify them and try them again to test your understanding of the language. To do that, you need a JavaScript interpreter.

The easiest way to try out a few lines of JavaScript is to open up the web developer tools in your web browser (with F12, Ctrl-Shift-I, or Command-Option-I) and select the Console tab. You can then type code at the prompt and see the results as you type. Browser developer tools often appear as panes at the bottom or right of the browser window, but you can usually detach them as separate windows (as pictured in Figure 1-1), which is often quite convenient.

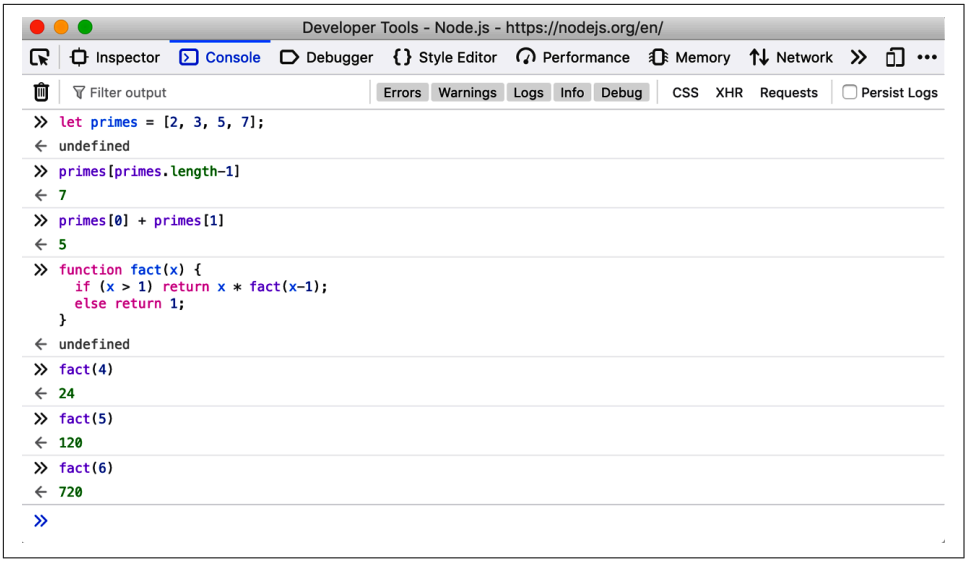

Figure 1-1. The JavaScript console in Firefox's Developer Tools

Another way to try out JavaScript code is to download and install Node from *https://nodejs.org*. Once Node is installed on your system, you can simply open a Terminal window and type **node** to begin an interactive JavaScript session like this one:

```
$ node
Welcome to Node.js v12.13.0.
Type ".help" for more information.
> .help
.break    Sometimes you get stuck, this gets you out
.clear    Alias for .break
.editor   Enter editor mode
.exit     Exit the repl
.help     Print this help message
.load     Load JS from a file into the REPL session
.save     Save all evaluated commands in this REPL session to a file

Press ^C to abort current expression, ^D to exit the repl
> let x = 2, y = 3;
undefined
> x + y
5
> (x === 2) && (y === 3)
true
> (x > 3) || (y < 3)
false
```

1.2 Hello World

When you are ready to start experimenting with longer chunks of code, these line-by-line interactive environments may no longer be suitable, and you will probably prefer to write your code in a text editor. From there, you can copy and paste to the JavaScript console or into a Node session. Or you can save your code to a file (the traditional filename extension for JavaScript code is *.js*) and then run that file of JavaScript code with Node:

```
$ node snippet.js
```

If you use Node in a noninteractive manner like this, it won't automatically print out the value of all the code you run, so you'll have to do that yourself. You can use the function `console.log()` to display text and other JavaScript values in your terminal window or in a browser's developer tools console. So, for example, if you create a *hello.js* file containing this line of code:

```
console.log("Hello World!");
```

and execute the file with `node hello.js`, you'll see the message "Hello World!" printed out.

If you want to see that same message printed out in the JavaScript console of a web browser, create a new file named *hello.html*, and put this text in it:

```
<script src="hello.js"></script>
```

Then load *hello.html* into your web browser using a `file://` URL like this one:

```
file:///Users/username/javascript/hello.html
```

Open the developer tools window to see the greeting in the console.

1.3 A Tour of JavaScript

This section presents a quick introduction, through code examples, to the JavaScript language. After this introductory chapter, we dive into JavaScript at the lowest level: Chapter 2 explains things like JavaScript comments, semicolons, and the Unicode character set. Chapter 3 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.

// A variable is a symbolic name for a value.
// Variables are declared with the let keyword:
let 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;                   // Numbers can be integers or reals.
x = "hello world";          // Strings of text in quotation marks.
x = 'JavaScript';           // Single quote marks also delimit strings.
x = true;                   // A Boolean value.
x = false;                  // The other Boolean value.
x = null;                   // Null is a special value that means "no value."
x = undefined;              // Undefined is another special value like null.
```

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

```
// JavaScript's most important datatype is the object.
// An object is a collection of name/value pairs, or a string to value map.
let book = {                // Objects are enclosed in curly braces.
    topic: "JavaScript",    // The property "topic" has value "JavaScript."
    edition: 7              // The property "edition" has value 7
};                          // The curly brace marks the end of the object.

// Access the properties of an object with . or []:
book.topic                  // => "JavaScript"
book["edition"]             // => 7: another way to access property values.
book.author = "Flanagan";   // Create new properties by assignment.
book.contents = {};         // {} is an empty object with no properties.

// Conditionally access properties with ?. (ES2020):
book.contents?.ch01?.sect1 // => undefined: book.contents has no ch01 property.

// JavaScript also supports arrays (numerically indexed lists) of values:
let 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.
let empty = [];            // [] is an empty array with no elements.
empty.length               // => 0

// Arrays and objects can hold other arrays and objects:
let points = [             // An array with 2 elements.
    {x: 0, y: 0},          // Each element is an object.
    {x: 1, y: 1}
];
let 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.
};
```

Comment Syntax in Code Examples

You may have noticed in the preceding code that some of the comments begin with an arrow (=>). These show the value produced by the code before the comment and are my attempt to emulate an interactive JavaScript environment like a web browser console in a printed book.

Those `// =>` comments also serve as an *assertion*, and I've written a tool that tests the code and verifies that it produces the value specified in the comment. This should help, I hope, to reduce errors in the book.

There are two related styles of comment/assertion. If you see a comment of the form `// a == 42`, it means that after the code before the comment runs, the variable `a` will have the value 42. If you see a comment of the form `// !`, it means that the code on the line before the comment throws an exception (and the rest of the comment after the exclamation mark usually explains what kind of exception is thrown).

You'll see these comments used throughout the book.

The syntax illustrated here 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. An *expression* is a phrase of JavaScript that can be *evaluated* to produce a value. For example, the use of `.` and `[]` to refer to the value of an object property or array element is an expression.

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

```
// Operators act on values (the operands) to produce a new value.
// Arithmetic operators are some of the simplest:
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 also work
"3" + "2"                   // => "32": + adds numbers, concatenates strings

// JavaScript defines some shorthand arithmetic operators
let 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.
let 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 JavaScript expressions are like phrases, then JavaScript *statements* are like full sentences. Statements are the topic of Chapter 5. 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, 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. You'll see examples 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, 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

let 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
```

In ES6 and later, there is a shorthand syntax for defining functions. This concise syntax uses `=>` to separate the argument list from the function body, so functions defined this way are known as *arrow functions*. Arrow functions are most commonly used when you want to pass an unnamed function as an argument to another function. The preceding code looks like this when rewritten to use arrow functions:

```
const plus1 = x => x + 1;   // The input x maps to the output x + 1
const square = x => x * x;  // The input x maps to the output x * x
```

```
plus1(y)                    // => 4: function invocation is the same
square(plus1(y))            // => 16
```

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

```
// When functions are assigned to the properties of an object, we call
// them "methods."  All JavaScript objects (including arrays) have methods:
let 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 earlier.
points.dist = function() { // Define a method to compute distance between points
    let p1 = this[0];       // First element of array we're invoked on
    let p2 = this[1];       // Second element of the "this" object
    let a = p2.x-p1.x;      // Difference in x coordinates
    let 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()               // => Math.sqrt(2): 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.
abs(-10) === abs(10)        // => true

function sum(array) {       // Compute the sum of the elements of an array
    let sum = 0;            // Start with an initial sum of 0.
    for(let x of array) {  // Loop over array, assigning each element to x.
        sum += x;           // Add the element value to the sum.
    }                       // This is the end of the loop.
    return sum;             // Return the sum.
}
sum(primes)                 // => 28: sum of the first 5 primes 2+3+5+7+11

function factorial(n) {     // A function to compute factorials
    let 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
    let 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 supports an object-oriented programming style, but it is significantly different than "classical" object-oriented programming languages. Chapter 9 covers object-oriented programming in JavaScript in detail, with lots of examples. 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 `distance()`, that computes the distance of the point from the origin:

```
class Point {                // By convention, class names are capitalized.
    constructor(x, y) {      // Constructor function to initialize new instances.
        this.x = x;          // This keyword is the new object being initialized.
        this.y = y;          // Store function arguments as object properties.
    }                        // No return is necessary in constructor functions.

    distance() {             // Method to compute distance from origin to point.
        return Math.sqrt(    // Return the square root of x² + y².
            this.x * this.x +  // this refers to the Point object on which
            this.y * this.y    // the distance method is invoked.
        );
    }
}

// Use the Point() constructor function with "new" to create Point objects
let p = new Point(1, 1);   // The geometric point (1,1).

// Now use a method of the Point object p
p.distance()               // => Math.SQRT2
```

This introductory tour of JavaScript's fundamental syntax and capabilities ends here, but the book continues with self-contained chapters that cover additional features of the language:

Chapter 10, Modules
: Shows how JavaScript code in one file or script can use JavaScript functions and classes defined in other files or scripts.

Chapter 11, The JavaScript Standard Library
: Covers the built-in functions and classes that are available to all JavaScript programs. This includes important data stuctures like maps and sets, a regular

expression class for textual pattern matching, functions for serializing JavaScript data structures, and much more.

Chapter 12, Iterators and Generators
: Explains how the `for/of` loop works and how you can make your own classes iterable with `for/of`. It also covers generator functions and the `yield` statement.

Chapter 13, Asynchronous JavaScript
: This chapter is an in-depth exploration of asynchronous programming in JavaScript, covering callbacks and events, Promise-based APIs, and the `async` and `await` keywords. Although the core JavaScript language is not asynchronous, asynchronous APIs are the default in both web browsers and Node, and this chapter explains the techniques for working with those APIs.

Chapter 14, Metaprogramming
: Introduces a number of advanced features of JavaScript that may be of interest to programmers writing libraries of code for other JavaScript programmers to use.

Chapter 15, JavaScript in Web Browsers
: Introduces the web browser host environment, explains how web browsers execute JavaScript code, and covers the most important of the many APIs defined by web browsers. This is by far the longest chapter in the book.

Chapter 16, Server-Side JavaScript with Node
: Introduces the Node host environment, covering the fundamental programming model and the data structures and APIs that are most important to understand.

Chapter 17, JavaScript Tools and Extensions
: Covers tools and language extensions that are worth knowing about because they are widely used and may make you a more productive programmer.

1.4 Example: Character Frequency Histograms

This chapter concludes with a short but nontrivial JavaScript program. Example 1-1 is a Node program that reads text from standard input, computes a character frequency histogram from that text, and then prints out the histogram. You could invoke the program like this to analyze the character frequency of its own source code:

```
$ node charfreq.js < charfreq.js
T: ########## 11.22%
E: ########## 10.15%
R: ####### 6.68%
S: ###### 6.44%
A: ###### 6.16%
N: ###### 5.81%
O: ##### 5.45%
I: ##### 4.54%
```

```
    H: #### 4.07%
    C: ### 3.36%
    L: ### 3.20%
    U: ### 3.08%
    /: ### 2.88%
```

This example uses a number of advanced JavaScript features and is intended to demonstrate what real-world JavaScript programs can look like. You should not expect to understand all of the code yet, but be assured that all of it will be explained in the chapters that follow.

Example 1-1. Computing character frequency histograms with JavaScript

```
/**
 * This Node program reads text from standard input, computes the frequency
 * of each letter in that text, and displays a histogram of the most
 * frequently used characters. It requires Node 12 or higher to run.
 *
 * In a Unix-type environment you can invoke the program like this:
 *    node charfreq.js < corpus.txt
 */

// This class extends Map so that the get() method returns the specified
// value instead of null when the key is not in the map
class DefaultMap extends Map {
    constructor(defaultValue) {
        super();                            // Invoke superclass constructor
        this.defaultValue = defaultValue; // Remember the default value
    }

    get(key) {
        if (this.has(key)) {              // If the key is already in the map
            return super.get(key);        // return its value from superclass.
        }
        else {
            return this.defaultValue;     // Otherwise return the default value
        }
    }
}

// This class computes and displays letter frequency histograms
class Histogram {
    constructor() {
        this.letterCounts = new DefaultMap(0);  // Map from letters to counts
        this.totalLetters = 0;                  // How many letters in all
    }

    // This function updates the histogram with the letters of text.
    add(text) {
        // Remove whitespace from the text, and convert to upper case
        text = text.replace(/\s/g, "").toUpperCase();
```

```
        // Now loop through the characters of the text
        for(let character of text) {
            let count = this.letterCounts.get(character); // Get old count
            this.letterCounts.set(character, count+1);    // Increment it
            this.totalLetters++;
        }
    }

    // Convert the histogram to a string that displays an ASCII graphic
    toString() {
        // Convert the Map to an array of [key,value] arrays
        let entries = [...this.letterCounts];

        // Sort the array by count, then alphabetically
        entries.sort((a,b) => {              // A function to define sort order.
            if (a[1] === b[1]) {             // If the counts are the same
                return a[0] < b[0] ? -1 : 1; // sort alphabetically.
            } else {                         // If the counts differ
                return b[1] - a[1];          // sort by largest count.
            }
        });

        // Convert the counts to percentages
        for(let entry of entries) {
            entry[1] = entry[1] / this.totalLetters*100;
        }

        // Drop any entries less than 1%
        entries = entries.filter(entry => entry[1] >= 1);

        // Now convert each entry to a line of text
        let lines = entries.map(
            ([l,n]) => `${l}: ${"#".repeat(Math.round(n))} ${n.toFixed(2)}%`
        );

        // And return the concatenated lines, separated by newline characters.
        return lines.join("\n");
    }
}

// This async (Promise-returning) function creates a Histogram object,
// asynchronously reads chunks of text from standard input, and adds those chunks to
// the histogram. When it reaches the end of the stream, it returns this histogram
async function histogramFromStdin() {
    process.stdin.setEncoding("utf-8"); // Read Unicode strings, not bytes
    let histogram = new Histogram();
    for await (let chunk of process.stdin) {
        histogram.add(chunk);
    }
    return histogram;
}
```

```
// This one final line of code is the main body of the program.
// It makes a Histogram object from standard input, then prints the histogram.
histogramFromStdin().then(histogram => { console.log(histogram.toString()); });
```

1.5 Summary

This book explains JavaScript from the bottom up. This means that we start with low-level details like comments, identifiers, variables, and types; then build to expressions, statements, objects, and functions; and then cover high-level language abstractions like classes and modules. I take the word *definitive* in the title of this book seriously, and the coming chapters explain the language at a level of detail that may feel off-putting at first. True mastery of JavaScript requires an understanding of the details, however, and I hope that you will make time to read this book cover to cover. But please don't feel that you need to do that on your first reading. If you find yourself feeling bogged down in a section, simply skip to the next. You can come back and master the details once you have a working knowledge of the language as a whole.

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 what variable names look like, the delimiter characters for comments, and how one program statement is separated from the next, for example. This short chapter documents the lexical structure of JavaScript. It covers:

- Case sensitivity, spaces, and line breaks
- Comments
- Literals
- Identifiers and reserved words
- Unicode
- Optional semicolons

2.1 The Text of a JavaScript Program

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.

JavaScript ignores spaces that appear between tokens in programs. For the most part, JavaScript also ignores line breaks (but see §2.6 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 tabs, assorted ASCII control characters, and various Unicode space characters as whitespace. JavaScript recognizes newlines, carriage returns, and a carriage return/line feed sequence as line terminators.

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 a multi-line comment. The extra * characters at the start of
 * each line are not a required part of the syntax; they just look cool!
 */
```

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
null            // Absence of an object
```

Complete details on numeric and string literals appear in Chapter 3.

2.4 Identifiers and Reserved Words

An *identifier* is simply a name. In JavaScript, identifiers are used to name constants, variables, properties, functions, and classes 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
```

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 in the next section.

2.4.1 Reserved Words

The following words are part of the JavaScript language. Many of these (such as `if`, `while`, and `for`) are reserved keywords that must not be used as the names of constants, variables, functions, or classes (though they can all be used as the names of properties within an object). Others (such as `from`, `of`, `get`, and `set`) are used in limited contexts with no syntactic ambiguity and are perfectly legal as identifiers. Still other keywords (such as `let`) can't be fully reserved in order to retain backward compatibility with older programs, and so there are complex rules that govern when they can be used as identifiers and when they cannot. (`let` can be used as a variable name if declared with `var` outside of a class, for example, but not if declared inside a class or with `const`.) The simplest course is to avoid using any of these words as identifiers, except for `from`, `set`, and `target`, which are safe to use and are already in common use.

```
as      const     export    get         null    target   void
async   continue  extends   if          of      this     while
await   debugger  false     import      return  throw    with
break   default   finally   in          set     true     yield
case    delete    for       instanceof  static  try
catch   do        from      let         super   typeof
class   else      function  new         switch  var
```

JavaScript also reserves or restricts the use of certain keywords that are not currently used by the language but that might be used in future versions:

```
enum  implements  interface  package  private  protected  public
```

For historical reasons, `arguments` and `eval` are not allowed as identifiers in certain circumstances and are best avoided entirely.

2.5 Unicode

JavaScript programs are written using the Unicode character set, and you can use any Unicode characters in strings and comments. For portability and ease of editing, it is common to use only ASCII letters and digits in identifiers. But this is a programming convention only, and the language allows Unicode letters, digits, and ideographs (but

not emojis) in identifiers. This means that programmers can use mathematical symbols and words from non-English languages as constants and variables:

```
const π = 3.14;
const sí = true;
```

2.5.1 Unicode Escape Sequences

Some computer hardware and software cannot display, input, or correctly process the full set of Unicode characters. To support programmers and systems using older technology, JavaScript defines escape sequences that allow us to write Unicode characters using only ASCII characters. These Unicode escapes begin with the characters `\u` and are either followed by exactly four hexadecimal digits (using uppercase or lowercase letters A–F) or by one to six hexadecimal digits enclosed within curly braces. These Unicode escapes may appear in JavaScript string literals, regular expression literals, and identifiers (but not in language keywords). The Unicode escape for the character "é," for example, is `\u00E9`; here are three different ways to write a variable name that includes this character:

```
let café = 1; // Define a variable using a Unicode character
caf\u00e9     // => 1; access the variable using an escape sequence
caf\u{E9}     // => 1; another form of the same escape sequence
```

Early versions of JavaScript only supported the four-digit escape sequence. The version with curly braces was introduced in ES6 to better support Unicode codepoints that require more than 16 bits, such as emoji:

```
console.log("\u{1F600}");  // Prints a smiley face emoji
```

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

2.5.2 Unicode Normalization

If you use non-ASCII characters in your JavaScript programs, you must be aware that 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 typically look exactly the same when displayed by a text editor, but they have different binary encodings, meaning that they are considered different by JavaScript, which can lead to very confusing programs:

```
const café = 1;  // This constant is named "caf\u{e9}"
const café = 2;  // This constant is different: "cafe\u{301}"
café  // => 1: this constant has one value
café  // => 2: this indistinguishable constant has a different value
```

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 does *not* do any normalization on its own. If you plan to use Unicode characters in your JavaScript programs, you should ensure that your editor or some other tool performs Unicode normalization of your source code to prevent you from ending up with different but visually indistinguishable identifiers.

2.6 Optional Semicolons

Like many programming languages, JavaScript uses the semicolon (`;`) to separate statements (see Chapter 5) from one another. This is important for making 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 adding an implicit semicolon. More formally (and with three exceptions described a bit later), 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:

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

JavaScript interprets this code like this:

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

JavaScript does treat the first line break as a semicolon because it cannot parse the code `let 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:

```
let 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:

```
let 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:

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

There are three 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`, `throw`, `yield`, `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;
```

This means 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. The third exception involves functions defined using concise "arrow" syntax: the => arrow itself must appear on the same line as the parameter list.

2.7 Summary

This chapter has shown how JavaScript programs are written at the lowest level. The next chapter takes us one step higher and introduces the primitive types and values (numbers, strings, and so on) that serve as the basic units of computation for JavaScript programs.

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. Variables have names, and they allow use of those names in our programs to refer to values. The way that variables work is another fundamental characteristic of any programming language. This chapter explains types, values, and variables in JavaScript. It begins with an overview and some definitions.

3.1 Overview and Definitions

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 (§3.2) and string (§3.3) types in JavaScript. Booleans are covered in §3.4.

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. §3.5 has more about `null` and `undefined`. ES6 adds a new special-purpose type, known as Symbol, that enables the definition of language extensions without harming backward compatibility. Symbols are covered briefly in §3.6.

Any JavaScript value that is not a number, a string, a boolean, a symbol, `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 or

another object). One very special object, the *global object*, is covered in §3.7, 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.

In addition to basic objects and arrays, JavaScript defines a number of other useful object types. A Set object represents a set of values. A Map object represents a mapping from keys to values. Various "typed array" types facilitate operations on arrays of bytes and other binary data. The RegExp type represents textual patterns and enables sophisticated matching, searching, and replacing operations on strings. The Date type represents dates and times and supports rudimentary date arithmetic. Error and its subtypes represent errors that can arise when executing JavaScript code. All of these types are covered in Chapter 11.

JavaScript differs from more static languages in that functions and classes are not just part of the language syntax: they are themselves values that can be manipulated by JavaScript programs. Like any JavaScript value that is not a primitive value, functions and classes are a specialized kind of object. They are covered in detail in Chapters 8 and 9.

The JavaScript interpreter performs automatic garbage collection for memory management. This means that a JavaScript programmer generally does not need to worry about destruction or deallocation of objects or other values. When a value 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 programmers do sometimes need to take care to ensure that values do not inadvertently remain reachable—and therefore nonreclaimable—longer than necessary.)

JavaScript supports an object-oriented programming style. 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, boolean, and symbol values behave as if they have methods. In JavaScript, `null` and `undefined` are the only values that methods cannot be invoked on.

JavaScript's object types are *mutable* and its primitive types are *immutable*. A value of a mutable type can change: a JavaScript program can change the values of object properties and array elements. Numbers, booleans, symbols, `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 §3.8.

JavaScript liberally converts values 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. And if you use a non-boolean value where a boolean is expected, JavaScript will convert accordingly. The rules for value conversion are explained in §3.9. JavaScript's liberal value conversion rules affect its definition of equality, and the `==` equality operator performs type conversions as described in §3.9.1. (In practice, however, the `==` equality operator is deprecated in favor of the strict equality operator `===`, which does no type conversions. See §4.9.1 for more about both operators.)

Constants and variables allow you to use names to refer to values in your programs. Constants are declared with `const` and variables are declared with `let` (or with `var` in older JavaScript code). JavaScript constants and variables are *untyped*: declarations do not specify what kind of values will be assigned. Variable declaration and assignment are covered in §3.10.

As you can see from this long introduction, this is a wide-ranging chapter that explains many fundamental details about how data is represented and manipulated in JavaScript. We'll begin by diving right in to the details of JavaScript numbers and text.

3.2 Numbers

JavaScript's primary numeric type, Number, is used to represent integers and to approximate real numbers. 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 −9,007,199,254,740,992 (-2^{53}) and 9,007,199,254,740,992 (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

1 This is the format for numbers of type `double` in Java, C++, and most modern programming languages.

operators described in Chapter 4) are performed with 32-bit integers. If you need to exactly represent larger integers, see §3.2.5.

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.

3.2.1 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       // => 255: (15*16 + 15)
0xBADCAFE  // => 195939070
```

In ES6 and later, you can also express integers in binary (base 2) or octal (base 8) using the prefixes `0b` and `0o` (or `0B` and `0O`) instead of `0x`:

```
0b10101  // => 21:  (1*16 + 0*8 + 1*4 + 0*2 + 1*1)
0o377    // => 255: (3*64 + 7*8 + 7*1)
```

3.2.2 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
2345.6789
.333333333333333333
6.02e23        // 6.02 × 10²³
1.4738223E-32  // 1.4738223 × 10⁻³²
```

Separators in Numeric Literals

You can use underscores within numeric literals to break long literals up into chunks that are easier to read:

```
let billion = 1_000_000_000;   // Underscore as a thousands separator.
let bytes = 0x89_AB_CD_EF;     // As a bytes separator.
let bits = 0b0001_1101_0111;   // As a nibble separator.
let fraction = 0.123_456_789;  // Works in the fractional part, too.
```

At the time of this writing in early 2020, underscores in numeric literals are not yet formally standardized as part of JavaScript. But they are in the advanced stages of the standardization process and are implemented by all major browsers and by Node.

3.2.3 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). ES2016 adds ** for exponentiation. 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)            // => 3**0.5: the square root of 3
Math.pow(3, 1/3)        // => 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
```

ES6 defines more functions on the `Math` object:

```
Math.cbrt(27)    // => 3: cube root
Math.hypot(3, 4) // => 5: square root of sum of squares of all arguments
Math.log10(100)  // => 2: Base-10 logarithm
Math.log2(1024)  // => 10: Base-2 logarithm
Math.log1p(x)    // Natural log of (1+x); accurate for very small x
Math.expm1(x)    // Math.exp(x)-1; the inverse of Math.log1p()
Math.sign(x)     // -1, 0, or 1 for arguments <, ==, or > 0
Math.imul(2,3)   // => 6: optimized multiplication of 32-bit integers
Math.clz32(0xf)  // => 28: number of leading zero bits in a 32-bit integer
Math.trunc(3.9)  // => 3: convert to an integer by truncating fractional part
Math.fround(x)   // Round to nearest 32-bit float number
Math.sinh(x)     // Hyperbolic sine. Also Math.cosh(), Math.tanh()
Math.asinh(x)    // Hyperbolic arcsine. Also Math.acosh(), Math.atanh()
```

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, `Infinity`. Similarly, when the absolute value of a negative value becomes larger than the absolute value of the largest representable negative number, the result is negative infinity, `-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, `NaN`. `NaN` also arises if you attempt to divide infinity by infinity, 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 constants `Infinity` and `NaN` to hold the positive infinity and not-a-number value, and these values are also available as properties of the `Number` object:

```
Infinity                 // A positive number too big to represent
Number.POSITIVE_INFINITY // Same value
1/0                      // => Infinity
Number.MAX_VALUE * 2     // => Infinity; overflow
```

```
-Infinity                  // A negative number too big to represent
Number.NEGATIVE_INFINITY   // The same value
-1/0                       // => -Infinity
-Number.MAX_VALUE * 2      // => -Infinity

NaN                        // The not-a-number value
Number.NaN                 // The same value, written another way
0/0                        // => NaN
Infinity/Infinity          // => NaN

Number.MIN_VALUE/2         // => 0: underflow
-Number.MIN_VALUE/2        // => -0: negative zero
-1/Infinity                // -> -0: also negative 0
-0

// The following Number properties are defined in ES6
Number.parseInt()       // Same as the global parseInt() function
Number.parseFloat()     // Same as the global parseFloat() function
Number.isNaN(x)         // Is x the NaN value?
Number.isFinite(x)      // Is x a number and finite?
Number.isInteger(x)     // Is x an integer?
Number.isSafeInteger(x) // Is x an integer -(2**53) < x < 2**53?
Number.MIN_SAFE_INTEGER // => -(2**53 - 1)
Number.MAX_SAFE_INTEGER // => 2**53 - 1
Number.EPSILON          // => 2**-52: smallest difference between numbers
```

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 must write `x != x` or `Number.isNaN(x)`. Those expressions will be true if, and only if, `x` has the same value as the global constant `NaN`.

The global function `isNaN()` is similar to `Number.isNaN()`. It returns `true` if its argument is `NaN`, or if that argument is a non-numeric value that cannot be converted to a number. The related function `Number.isFinite()` returns `true` if its argument is a number other than `NaN`, `Infinity`, or `-Infinity`. The global `isFinite()` function returns `true` if its argument is, or can be converted to, a finite number.

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:

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

3.2.4 Binary Floating-Point and Rounding Errors

There are infinitely many real numbers, but only a finite number of them (18,437,736,874,454,810,627, 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:

```
let x = .3 - .2;    // thirty cents minus 20 cents
let 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 shown here are *very* close to each other and to the correct value. The computed values are adequate for almost any purpose; the problem only arises when we attempt to compare values for equality.

If these floating-point approximations are problematic for your programs, consider using scaled integers. For example, you might manipulate monetary values as integer cents rather than fractional dollars.

3.2.5 Arbitrary Precision Integers with BigInt

One of the newest features of JavaScript, defined in ES2020, is a new numeric type known as BigInt. As of early 2020, it has been implemented in Chrome, Firefox, Edge, and Node, and there is an implementation in progress in Safari. As the name implies, BigInt is a numeric type whose values are integers. The type was added to JavaScript mainly to allow the representation of 64-bit integers, which are required for compatibility with many other programming languages and APIs. But BigInt values can have thousands or even millions of digits, should you have need to work with numbers

that large. (Note, however, that BigInt implementations are not suitable for cryptography because they do not attempt to prevent timing attacks.)

BigInt literals are written as a string of digits followed by a lowercase letter `n`. By default, the are in base 10, but you can use the `0b`, `0o`, and `0x` prefixes for binary, octal, and hexadecimal BigInts:

```
1234n               // A not-so-big BigInt literal
0b111111n           // A binary BigInt
0o7777n             // An octal BigInt
0x8000000000000000n // => 2n**63n: A 64-bit integer
```

You can use `BigInt()` as a function for converting regular JavaScript numbers or strings to BigInt values:

```
BigInt(Number.MAX_SAFE_INTEGER)     // => 9007199254740991n
let string = "1" + "0".repeat(100); // 1 followed by 100 zeros.
BigInt(string)                      // => 10n**100n: one googol
```

Arithmetic with BigInt values works like arithmetic with regular JavaScript numbers, except that division drops any remainder and rounds down (toward zero):

```
1000n + 2000n  // => 3000n
3000n - 2000n  // => 1000n
2000n * 3000n  // => 6000000n
3000n / 997n   // => 3n: the quotient is 3
3000n % 997n   // => 9n: and the remainder is 9
(2n ** 131071n) - 1n  // A Mersenne prime with 39457 decimal digits
```

Although the standard `+`, `-`, `*`, `/`, `%`, and `**` operators work with BigInt, it is important to understand that you may not mix operands of type BigInt with regular number operands. This may seem confusing at first, but there is a good reason for it. If one numeric type was more general than the other, it would be easy to define arithmetic on mixed operands to simply return a value of the more general type. But neither type is more general than the other: BigInt can represent extraordinarily large values, making it more general than regular numbers. But BigInt can only represent integers, making the regular JavaScript number type more general. There is no way around this problem, so JavaScript sidesteps it by simply not allowing mixed operands to the arithmetic operators.

Comparison operators, by contrast, do work with mixed numeric types (but see §3.9.1 for more about the difference between == and ===):

```
1 < 2n     // => true
2 > 1n     // => true
0 == 0n    // => true
0 === 0n   // => false: the === checks for type equality as well
```

The bitwise operators (described in §4.8.3) generally work with BigInt operands. None of the functions of the `Math` object accept BigInt operands, however.

3.2.6 Dates and Times

JavaScript defines a simple Date class for representing and manipulating the numbers that represent dates and times. JavaScript Dates are objects, but they also have a numeric representation as a *timestamp* that specifies the number of elapsed milliseconds since January 1, 1970:

```
let timestamp = Date.now();  // The current time as a timestamp (a number).
let now = new Date();        // The current time as a Date object.
let ms = now.getTime();      // Convert to a millisecond timestamp.
let iso = now.toISOString(); // Convert to a string in standard format.
```

The Date class and its methods are covered in detail in §11.4. But we will see Date objects again in §3.9.3 when we examine the details of JavaScript type conversions.

3.3 Text

The JavaScript type for representing text is the *string*. A string is an immutable ordered sequence of 16-bit values, each of which typically represents a Unicode character. 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 one element of a string. Unicode characters whose codepoints do not fit in 16 bits are encoded using 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:

```
let euro = "€";
let love = "❤";
euro.length   // => 1: this character has one 16-bit element
love.length   // => 2: UTF-16 encoding of ❤ is "\ud83d\udc99"
```

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

In ES6, however, strings are *iterable*, and if you use the `for/of` loop or `...` operator with a string, it will iterate the actual characters of the string, not the 16-bit values.

3.3.1 String Literals

To include a string in a JavaScript program, simply enclose the characters of the string within a matched pair of single or double quotes or backticks (' or " or `). Double-quote characters and backticks may be contained within strings delimited by single-quote characters, and similarly for strings delimited by double quotes and backticks. 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?"
"τ is the ratio of a circle's circumference to its radius"
`"She said 'hi'", he said.`
```

Strings delimited with backticks are a feature of ES6, and allow JavaScript expressions to be embedded within (or *interpolated* into) the string literal. This expression interpolation syntax is covered in §3.3.4.

The original versions of JavaScript required string literals to be written on a single line, and it is common to see JavaScript code that creates long strings by concatenating single-line strings with the + operator. As of ES5, 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 single-quoted or double-quoted string literal, use the character sequence \n (documented in the next section). The ES6 backtick syntax allows strings to be broken across multiple lines, and in this case, the line terminators are part of the string literal:

```
// A string representing 2 lines written on one line:
'two\nlines'

// A one-line string written on 3 lines:
"one\
 long\
 line"

// A two-line string written on two lines:
`the newline character at the end of this line
is included literally in this string`
```

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>
```

3.3.2 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 earlier, 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. Three escape sequences are generic and can be used to represent any character by specifying its Unicode character code as a hexadecimal number. For example, the sequence `\xA9` represents the copyright symbol, which has the Unicode encoding given by the hexadecimal number A9. Similarly, the `\u` escape represents an arbitrary Unicode character specified by four hexadecimal digits or one to five digits when the digits are enclosed in curly braces: `\u03c0` represents the character π, for example, and `\u{1f600}` represents the "grinning face" emoji.

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`)

Sequence	Character represented
\r	Carriage return (\u000D)
\"	Double quote (\u0022)
\'	Apostrophe or single quote (\u0027)
\\	Backslash (\u005C)
\x*nn*	The Unicode character specified by the two hexadecimal digits *nn*
\u*nnnn*	The Unicode character specified by the four hexadecimal digits *nnnn*
\u{*n*}	The Unicode character specified by the codepoint *n*, where *n* is one to six hexadecimal digits between 0 and 10FFFF (ES6)

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 earlier, ES5 allows a backslash before a line break to break a string literal across multiple lines.

3.3.3 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:

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

Strings can be compared with the standard === equality and !== inequality operators: two strings are equal if and only if they consist of exactly the same sequence of 16-bit values. Strings can also be compared with the <, <=, >, and >= operators. String comparison is done simply by comparing the 16-bit values. (For more robust locale-aware string comparison and sorting, see §11.7.3.)

To determine the length of a string—the number of 16-bit values it contains—use the length property of the string:

```
s.length
```

In addition to this length property, JavaScript provides a rich API for working with strings:

```
let s = "Hello, world"; // Start with some text.

// Obtaining portions of a string
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.split(", ")           // => ["Hello", "world"]: split at delimiter string
```

```
// Searching a string
s.indexOf("l")          // => 2: position of first letter l
s.indexOf("l", 3)       // => 3: position of first "l" at or after 3
s.indexOf("zz")         // => -1: s does not include the substring "zz"
s.lastIndexOf("l")      // => 10: position of last letter l

// Boolean searching functions in ES6 and later
s.startsWith("Hell")    // => true: the string starts with these
s.endsWith("!")         // => false: s does not end with that
s.includes("or")        // => true: s includes substring "or"

// Creating modified versions of a string
s.replace("llo", "ya")  // => "Heya, world"
s.toLowerCase()         // => "hello, world"
s.toUpperCase()         // => "HELLO, WORLD"
s.normalize()           // Unicode NFC normalization: ES6
s.normalize("NFD")      // NFD normalization. Also "NFKC", "NFKD"

// Inspecting individual (16-bit) characters of a string
s.charAt(0)             // => "H": the first character
s.charAt(s.length-1)    // => "d": the last character
s.charCodeAt(0)         // => 72: 16-bit number at the specified position
s.codePointAt(0)        // => 72: ES6, works for codepoints > 16 bits

// String padding functions in ES2017
"x".padStart(3)         // => "  x": add spaces on the left to a length of 3
"x".padEnd(3)           // => "x  ": add spaces on the right to a length of 3
"x".padStart(3, "*")    // => "**x": add stars on the left to a length of 3
"x".padEnd(3, "-")      // => "x--": add dashes on the right to a length of 3

// Space trimming functions. trim() is ES5; others ES2019
" test ".trim()         // => "test": remove spaces at start and end
" test ".trimStart()    // => "test ": remove spaces on left. Also trimLeft
" test ".trimEnd()      // => " test": remove spaces at right. Also trimRight

// Miscellaneous string methods
s.concat("!")           // => "Hello, world!": just use + operator instead
"<>".repeat(5)          // => "<><><><><>": concatenate n copies. ES6
```

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.

Strings can also 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:

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

3.3.4 Template Literals

In ES6 and later, string literals can be delimited with backticks:

```
let s = `hello world`;
```

This is more than just another string literal syntax, however, because these *template literals* can include arbitrary JavaScript expressions. The final value of a string literal in backticks is computed by evaluating any included expressions, converting the values of those expressions to strings and combining those computed strings with the literal characters within the backticks:

```
let name = "Bill";
let greeting = `Hello ${ name }.`;  // greeting == "Hello Bill."
```

Everything between the `${` and the matching `}` is interpreted as a JavaScript expression. Everything outside the curly braces is normal string literal text. The expression inside the braces is evaluated and then converted to a string and inserted into the template, replacing the dollar sign, the curly braces, and everything in between them.

A template literal may include any number of expressions. It can use any of the escape characters that normal strings can, and it can span any number of lines, with no special escaping required. The following template literal includes four JavaScript expressions, a Unicode escape sequence, and at least four newlines (the expression values may include newlines as well):

```
let errorMessage = `\
\u2718 Test failure at ${filename}:${linenumber}:
${exception.message}
Stack trace:
${exception.stack}
`;
```

The backslash at the end of the first line here escapes the initial newline so that the resulting string begins with the Unicode ✘ character (`\u2718`) rather than a newline.

Tagged template literals

A powerful but less commonly used feature of template literals is that, if a function name (or "tag") comes right before the opening backtick, then the text and the values of the expressions within the template literal are passed to the function. The value of this "tagged template literal" is the return value of the function. This could be used, for example, to apply HTML or SQL escaping to the values before substituting them into the text.

ES6 has one built-in tag function: `String.raw()`. It returns the text within backticks without any processing of backslash escapes:

```
`\n`.length            // => 1: the string has a single newline character
String.raw`\n`.length  // => 2: a backslash character and the letter n
```

Note that even though the tag portion of a tagged template literal is a function, there are no parentheses used in its invocation. In this very specific case, the backtick characters replace the open and close parentheses.

The ability to define your own template tag functions is a powerful feature of JavaScript. These functions do not need to return strings, and they can be used like constructors, as if defining a new literal syntax for the language. We'll see an example in §14.5.

3.3.5 Pattern Matching

JavaScript defines a datatype known as a *regular expression* (or RegExp) for describing and matching patterns in strings of text. RegExps are not one of the fundamental datatypes in JavaScript, but they have a literal syntax like numbers and strings do, so they sometimes seem like they are fundamental. The grammar of regular expression literals is complex and the API they define is nontrivial. They are documented in detail in §11.3. Because RegExps are powerful and commonly used for text processing, however, this section provides a brief overview.

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 nonzero 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:

```
let text = "testing: 1, 2, 3";   // Sample text
let 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 nondigits
```

3.4 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 §3.9, 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.

3.5 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`, `nil`, or `None`.

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 the return value of functions that do not explicitly return a value and the value of function parameters for which no argument is passed. `undefined` is a predefined global constant (not a language keyword like `null`, though this is not an important distinction in practice) that is initialized to the `undefined` value. 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.

I consider `undefined` to represent a system-level, unexpected, or error-like absence of value and `null` to represent a program-level, normal, or expected absence of value. I avoid using `null` and `undefined` when I can, but if I need to assign one of these values to a variable or property or pass or return one of these values to or from a function, I usually use `null`. Some programmers strive to avoid `null` entirely and use `undefined` in its place wherever they can.

3.6 Symbols

Symbols were introduced in ES6 to serve as non-string property names. To understand Symbols, you need to know that JavaScript's fundamental Object type is an unordered collection of properties, where each property has a name and a value. Property names are typically (and until ES6, were exclusively) strings. But in ES6 and later, Symbols can also serve this purpose:

```
let strname = "string name";      // A string to use as a property name
let symname = Symbol("propname"); // A Symbol to use as a property name
typeof strname                    // => "string": strname is a string
typeof symname                    // => "symbol": symname is a symbol
let o = {};                       // Create a new object
o[strname] = 1;                   // Define a property with a string name
o[symname] = 2;                   // Define a property with a Symbol name
o[strname]                        // => 1: access the string-named property
o[symname]                        // => 2: access the symbol-named property
```

The Symbol type does not have a literal syntax. To obtain a Symbol value, you call the `Symbol()` function. This function never returns the same value twice, even when called with the same argument. This means that if you call `Symbol()` to obtain a Symbol value, you can safely use that value as a property name to add a new property to an object and do not need to worry that you might be overwriting an existing property with the same name. Similarly, if you use symbolic property names and do not share those symbols, you can be confident that other modules of code in your program will not accidentally overwrite your properties.

In practice, Symbols serve as a language extension mechanism. When ES6 introduced the `for/of` loop (§5.4.4) and iterable objects (Chapter 12), it needed to define standard method that classes could implement to make themselves iterable. But standardizing any particular string name for this iterator method would have broken existing code, so a symbolic name was used instead. As we'll see in Chapter 12, `Symbol.iterator` is a Symbol value that can be used as a method name to make an object iterable.

The `Symbol()` function takes an optional string argument and returns a unique Symbol value. If you supply a string argument, that string will be included in the output

of the Symbol's `toString()` method. Note, however, that calling `Symbol()` twice with the same string produces two completely different Symbol values.

```
let s = Symbol("sym_x");
s.toString()            // => "Symbol(sym_x)"
```

`toString()` is the only interesting method of Symbol instances. There are two other Symbol-related functions you should know about, however. Sometimes when using Symbols, you want to keep them private to your own code so you have a guarantee that your properties will never conflict with properties used by other code. Other times, however, you might want to define a Symbol value and share it widely with other code. This would be the case, for example, if you were defining some kind of extension that you wanted other code to be able to participate in, as with the `Symbol.iterator` mechanism described earlier.

To serve this latter use case, JavaScript defines a global Symbol registry. The `Symbol.for()` function takes a string argument and returns a Symbol value that is associated with the string you pass. If no Symbol is already associated with that string, then a new one is created and returned; otherwise, the already existing Symbol is returned. That is, the `Symbol.for()` function is completely different than the `Symbol()` function: `Symbol()` never returns the same value twice, but `Symbol.for()` always returns the same value when called with the same string. The string passed to `Symbol.for()` appears in the output of `toString()` for the returned Symbol, and it can also be retrieved by calling `Symbol.keyFor()` on the returned Symbol.

```
let s = Symbol.for("shared");
let t = Symbol.for("shared");
s === t          // => true
s.toString()     // => "Symbol(shared)"
Symbol.keyFor(t) // => "shared"
```

3.7 The Global Object

The preceding sections 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 identifiers 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 constants like `undefined`, `Infinity`, and `NaN`
- Global functions like `isNaN()`, `parseInt()` (§3.9.2), and `eval()` (§4.12)

- Constructor functions like `Date()`, `RegExp()`, `String()`, `Object()`, and `Array()` (§3.9.2)
- Global objects like Math and JSON (§6.8)

The initial properties of the global object are not reserved words, but they deserve to be treated as if they are. This chapter has already described some of these global properties. Most of the others will be covered elsewhere in this book.

In Node, the global object has a property named `global` whose value is the global object itself, so you can always refer to the global object by the name `global` in Node programs.

In web browsers, 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 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. Web worker threads (§15.13) have a different global object than the Window with which they are associated. Code in a worker can refer to its global object as `self`.

ES2020 finally defines `globalThis` as the standard way to refer to the global object in any context. As of early 2020, this feature has been implemented by all modern browsers and by Node.

3.8 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:

```
let 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:

```
let 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

let 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 distinct objects are not equal even if they have the same properties and values. And two distinct arrays are not equal even if they have the same elements in the same order:

```
let o = {x: 1}, p = {x: 1};  // Two objects with the same properties
o === p                      // => false: distinct objects are never equal
let 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.

```
let a = [];   // The variable a refers to an empty array.
let 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 this code, 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 (§5.4.3):

```
let a = ["a","b","c"];              // An array we want to copy
let b = [];                         // A distinct array we'll copy into
for(let i = 0; i < a.length; i++) { // For each index of a[]
    b[i] = a[i];                    // Copy an element of a into b
}
let c = Array.from(b);              // In ES6, copy arrays with Array.from()
```

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 === b) return true;                // Identical arrays are equal
    if (a.length !== b.length) return false; // Different-size arrays not equal
    for(let 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
}
```

3.9 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
let n = 1 - "x";    // n == 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	to String	to Number	to Boolean
`undefined`	`"undefined"`	`NaN`	`false`
`null`	`"null"`	**`0`**	`false`
`true`	`"true"`	**`1`**	
`false`	`"false"`	**`0`**	
`""` (empty string)		**`0`**	**`false`**
`"1.2"` (nonempty, numeric)		`1.2`	`true`
`"one"` (nonempty, non-numeric)		`NaN`	`true`
`0`	`"0"`		**`false`**
`-0`	`"0"`		**`false`**
`1` (finite, non-zero)	`"1"`		`true`
`Infinity`	`"Infinity"`		`true`
`-Infinity`	`"-Infinity"`		`true`
`NaN`	`"NaN"`		**`false`**
`{}` (any object)	*see §3.9.3*	*see §3.9.3*	`true`
`[]` (empty array)	`""`	**`0`**	`true`
`[9]` (one numeric element)	`"9"`	**`9`**	`true`

Value	to String	to Number	to Boolean
`['a']` (any other array)	*use join() method*	`NaN`	`true`
`function(){}` (any function)	*see §3.9.3*	`NaN`	`true`

The primitive-to-primitive conversions shown in the table are relatively straightforward. Conversion to boolean was already discussed in §3.4. 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.

Object-to-primitive conversion is somewhat more complicated, and it is the subject of §3.9.3.

3.9.1 Conversions and Equality

JavaScript has two operators that test whether two values are equal. The "strict equality operator," `===`, does not consider its operands to be equal if they are not of the same type, and this is almost always the right operator to use when coding. But because JavaScript is so flexible with type conversions, it also defines the `==` operator with a flexible definition of equality. All of the following comparisons are true, for example:

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

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.

3.9.2 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()`, and `String()` functions:

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

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.

As an aside, note that the `Boolean()`, `Number()`, and `String()` functions can also be invoked—with `new`—as constructor. If you use them this way, you'll get a "wrapper" object that behaves just like a primitive boolean, number, or string value. These wrapper objects are a historical leftover from the earliest days of JavaScript, and there is never really any good reason to use them.

Certain JavaScript operators perform implicit type conversions and are sometimes used explicitly for the purpose 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 + ""   // => String(x)
+x       // => Number(x)
x-0      // => Number(x)
!!x      // => 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:

```
let n = 17;
let binary = "0b" + n.toString(2);  // binary == "0b10001"
let octal = "0o" + n.toString(8);   // octal == "0o21"
let hex = "0x" + n.toString(16);    // hex == "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:

```
let 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"
```

In addition to the number-formatting methods shown here, the Intl.NumberFormat class defines a more general, internationalized number-formatting method. See §11.7.1 for details.

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. Both `parseInt()` and `parseFloat()` skip leading whitespace, 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)
```

3.9.3 Object to Primitive Conversions

The previous sections have explained how you can explicitly convert values of one type to another type and have explained JavaScript's implicit conversions of values from one primitive type to another primitive type. This section covers the complicated rules that JavaScript uses to convert objects to primitive values. It is long and obscure, and if this is your first reading of this chapter, you should feel free to skip ahead to §3.10.

One reason for the complexity of JavaScript's object-to-primitive conversions is that some types of objects have more than one primitive representation. Date objects, for example, can be represented as strings or as numeric timestamps. The JavaScript specification defines three fundamental algorithms for converting objects to primitive values:

prefer-string
: This algorithm returns a primitive value, preferring a string value, if a conversion to string is possible.

prefer-number
: This algorithm returns a primitive value, preferring a number, if such a conversion is possible.

no-preference
: This algorithm expresses no preference about what type of primitive value is desired, and classes can define their own conversions. Of the built-in JavaScript types, all except Date implement this algorithm as *prefer-number*. The Date class implements this algorithm as *prefer-string*.

The implementation of these object-to-primitive conversion algorithms is explained at the end of this section. First, however, we explain how the algorithms are used in JavaScript.

Object-to-boolean conversions

Object-to-boolean conversions are trivial: all objects convert to `true`. Notice that this conversion does not require the use of the object-to-primitive algorithms described, and that it literally applies to *all* objects, including empty arrays and even the wrapper object `new Boolean(false)`.

Object-to-string conversions

When an object needs to be converted to a string, JavaScript first converts it to a primitive using the *prefer-string* algorithm, then converts the resulting primitive value to a string, if necessary, following the rules in Table 3-2.

This kind of conversion happens, for example, if you pass an object to a built-in function that expects a string argument, if you call `String()` as a conversion function, and when you interpolate objects into template literals (§3.3.4).

Object-to-number conversions

When an object needs to be converted to a number, JavaScript first converts it to a primitive value using the *prefer-number* algorithm, then converts the resulting primitive value to a number, if necessary, following the rules in Table 3-2.

Built-in JavaScript functions and methods that expect numeric arguments convert object arguments to numbers in this way, and most (see the exceptions that follow) JavaScript operators that expect numeric operands convert objects to numbers in this way as well.

Special case operator conversions

Operators are covered in detail in Chapter 4. Here, we explain the special case operators that do not use the basic object-to-string and object-to-number conversions described earlier.

The `+` operator in JavaScript performs numeric addition and string concatenation. If either of its operands is an object, JavaScript converts them to primitive values using the *no-preference* algorithm. Once it has two primitive values, it checks their types. If either argument is a string, it converts the other to a string and concatenates the strings. Otherwise, it converts both arguments to numbers and adds them.

The `==` and `!=` operators perform equality and inequality testing in a loose way that allows type conversions. If one operand is an object and the other is a primitive value, these operators convert the object to primitive using the *no-preference* algorithm and then compare the two primitive values.

Finally, the relational operators `<`, `<=`, `>`, and `>=` compare the order of their operands and can be used to compare both numbers and strings. If either operand is an object, it is converted to a primitive value using the *prefer-number* algorithm. Note, however, that unlike the object-to-number conversion, the primitive values returned by the *prefer-number* conversion are not then converted to numbers.

Note that the numeric representation of Date objects is meaningfully comparable with `<` and `>`, but the string representation is not. For Date objects, the *no-preference* algorithm converts to a string, so the fact that JavaScript uses the *prefer-number*

algorithm for these operators means that we can use them to compare the order of two Date objects.

The toString() and valueOf() methods

All objects inherit two conversion methods that are used by object-to-primitive conversions, and before we can explain the *prefer-string*, *prefer-number*, and *no-preference* conversion algorithms, we have to explain these two methods.

The first method is `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 §14.4.3):

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

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 converts 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) { f(x); }"
/\d+/g.toString()                  // => "/\\d+/g"
let d = new Date(2020,0,1);
d.toString()  // => "Wed Jan 01 2020 00:00:00 GMT-0800 (Pacific Standard Time)"
```

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 such as String, Number, and Boolean define `valueOf()` methods that simply 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:

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

Object-to-primitive conversion algorithms

With the `toString()` and `valueOf()` methods explained, we can now explain approximately how the three object-to-primitive algorithms work (the complete details are deferred until §14.4.7):

- The *prefer-string* algorithm first tries the `toString()` method. If the method is defined and returns a primitive value, then JavaScript uses that primitive value (even if it is not a string!). If `toString()` does not exist or if it returns an object, then JavaScript tries the `valueOf()` method. If that method exists and returns a primitive value, then JavaScript uses that value. Otherwise, the conversion fails with a TypeError.
- The *prefer-number* algorithm works like the *prefer-string* algorithm, except that it tries `valueOf()` first and `toString()` second.
- The *no-preference* algorithm depends on the class of the object being converted. If the object is a Date object, then JavaScript uses the *prefer-string* algorithm. For any other object, JavaScript uses the *prefer-number* algorithm.

The rules described here are true for all built-in JavaScript types and are the default rules for any classes you define yourself. §14.4.7 explains how you can define your own object-to-primitive conversion algorithms for the classes you define.

Before we leave this topic, it is worth noting that the details of the *prefer-number* conversion explain why empty arrays convert to the number 0 and single-element arrays can also convert to numbers:

```
Number([])    // => 0: this is unexpected!
Number([99])  // => 99: really?
```

The object-to-number conversion first converts the object to a primitive using the *prefer-number* algorithm, then converts the resulting primitive value to a number. The *prefer-number* algorithm tries `valueOf()` first and then falls back on `toString()`. But the Array class inherits the default `valueOf()` method, which does not return a primitive value. So when we try to convert an array to a number, we end up invoking the `toString()` method of the array. 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.

3.10 Variable Declaration and Assignment

One of the most fundamental techniques of computer programming is the use of names—or *identifiers*—to represent values. Binding a name to a value gives us a way to refer to that value and use it in the programs we write. When we do this, we typically say that we are assigning a value to a *variable*. The term "variable" implies that new values can be assigned: that the value associated with the variable may vary as our program runs. If we permanently assign a value to a name, then we call that name a *constant* instead of a variable.

Before you can use a variable or constant in a JavaScript program, you must *declare* it. In ES6 and later, this is done with the `let` and `const` keywords, which we explain next. Prior to ES6, variables were declared with `var`, which is more idiosyncratic and is explained later on in this section.

3.10.1 Declarations with let and const

In modern JavaScript (ES6 and later), variables are declared with the `let` keyword, like this:

```
let i;
let sum;
```

You can also declare multiple variables in a single `let` statement:

```
let i, sum;
```

It is a good programming practice to assign an initial value to your variables when you declare them, when this is possible:

```
let message = "hello";
let i = 0, j = 0, k = 0;
let x = 2, y = x*x; // Initializers can use previously declared variables
```

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

To declare a constant instead of a variable, use `const` instead of `let`. `const` works just like `let` except that you must initialize the constant when you declare it:

```
const H0 = 74;         // Hubble constant (km/s/Mpc)
const C = 299792.458;  // Speed of light in a vacuum (km/s)
const AU = 1.496E8;    // Astronomical Unit: distance to the sun (km)
```

As the name implies, constants cannot have their values changed, and any attempt to do so causes a TypeError to be thrown.

It is a common (but not universal) convention to declare constants using names with all capital letters such as `H0` or `HTTP_NOT_FOUND` as a way to distinguish them from variables.

When to Use const

There are two schools of thought about the use of the `const` keyword. One approach is to use `const` only for values that are fundamentally unchanging, like the physical constants shown, or program version numbers, or byte sequences used to identify file types, for example. Another approach recognizes that many of the so-called variables in our program don't actually ever change as our program runs. In this approach, we declare everything with `const`, and then if we find that we do actually want to allow the value to vary, we switch the declaration to `let`. This may help prevent bugs by ruling out accidental changes to variables that we did not intend.

In one approach, we use `const` only for values that *must not* change. In the other, we use `const` for any value that does not happen to change. I prefer the former approach in my own code.

In Chapter 5, we'll learn about the `for`, `for/in`, and `for/of` loop statements in JavaScript. Each of these loops includes a loop variable that gets a new value assigned to it on each iteration of the loop. JavaScript allows us to declare the loop variable as part of the loop syntax itself, and this is another common way to use `let`:

```
for(let i = 0, len = data.length; i < len; i++) console.log(data[i]);
for(let datum of data) console.log(datum);
for(let property in object) console.log(property);
```

It may seem surprising, but you can also use `const` to declare the loop "variables" for `for/in` and `for/of` loops, as long as the body of the loop does not reassign a new value. In this case, the `const` declaration is just saying that the value is constant for the duration of one loop iteration:

```
for(const datum of data) console.log(datum);
for(const property in object) console.log(property);
```

Variable and constant scope

The *scope* of a variable is the region of your program source code in which it is defined. Variables and constants declared with `let` and `const` are *block scoped*. This means that they are only defined within the block of code in which the `let` or `const` statement appears. JavaScript class and function definitions are blocks, and so are the bodies of `if/else` statements, `while` loops, `for` loops, and so on. Roughly speaking, if a variable or constant is declared within a set of curly braces, then those curly braces delimit the region of code in which the variable or constant is defined (though of course it is not legal to reference a variable or constant from lines of code that execute before the `let` or `const` statement that declares the variable). Variables and constants

declared as part of a `for`, `for/in`, or `for/of` loop have the loop body as their scope, even though they technically appear outside of the curly braces.

When a declaration appears at the top level, outside of any code blocks, we say it is a *global* variable or constant and has global scope. In Node and in client-side JavaScript modules (see Chapter 10), the scope of a global variable is the file that it is defined in. In traditional client-side JavaScript, however, the scope of a global variable is the HTML document in which it is defined. That is: if one `<script>` declares a global variable or constant, that variable or constant is defined in all of the `<script>` elements in that document (or at least all of the scripts that execute after the `let` or `const` statement executes).

Repeated declarations

It is a syntax error to use the same name with more than one `let` or `const` declaration in the same scope. It is legal (though a practice best avoided) to declare a new variable with the same name in a nested scope:

```
const x = 1;        // Declare x as a global constant
if (x === 1) {
    let x = 2;      // Inside a block x can refer to a different value
    console.log(x); // Prints 2
}
console.log(x);     // Prints 1: we're back in the global scope now
let x = 3;          // ERROR! Syntax error trying to re-declare x
```

Declarations and types

If you're used to statically typed languages such as C or Java, you may think that the primary purpose of variable declarations is to specify the type of values that may be assigned to a variable. But, as you have seen, there is no type associated with JavaScript's variable declarations.[2] A JavaScript variable can hold a value of any type. For example, it is perfectly legal (but generally poor programming style) in JavaScript to assign a number to a variable and then later assign a string to that variable:

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

3.10.2 Variable Declarations with var

In versions of JavaScript before ES6, the only way to declare a variable is with the `var` keyword, and there is no way to declare constants. The syntax of `var` is just like the syntax of `let`:

2 There are JavaScript extensions, such as TypeScript and Flow (§17.8), that allow types to be specified as part of variable declarations with syntax like `let x: number = 0;`.

```
var x;
var data = [], count = data.length;
for(var i = 0; i < count; i++) console.log(data[i]);
```

Although `var` and `let` have the same syntax, there are important differences in the way they work:

- Variables declared with `var` do not have block scope. Instead, they are scoped to the body of the containing function no matter how deeply nested they are inside that function.
- If you use `var` outside of a function body, it declares a global variable. But global variables declared with `var` differ from globals declared with `let` in an important way. Globals declared with `var` are implemented as properties of the global object (§3.7). The global object can be referenced as `globalThis`. So if you write `var x = 2;` outside of a function, it is like you wrote `globalThis.x = 2;`. Note however, that the analogy is not perfect: the properties created with global `var` declarations cannot be deleted with the `delete` operator (§4.13.4). Global variables and constants declared with `let` and `const` are not properties of the global object.
- Unlike variables declared with `let`, it is legal to declare the same variable multiple times with `var`. And because `var` variables have function scope instead of block scope, it is actually common to do this kind of redeclaration. The variable `i` is frequently used for integer values, and especially as the index variable of `for` loops. In a function with multiple `for` loops, it is typical for each one to begin `for(var i = 0; ...`. Because `var` does not scope these variables to the loop body, each of these loops is (harmlessly) re-declaring and re-initializing the same variable.
- One of the most unusual features of `var` declarations is known as *hoisting*. When a variable is declared with `var`, the declaration is lifted up (or "hoisted") to the top of the enclosing function. The initialization of the variable remains where you wrote it, but the definition of the variable moves to the top of the function. So variables declared with `var` can be used, without error, anywhere in the enclosing function. If the initialization code has not run yet, then the value of the variable may be `undefined`, but you won't get an error if you use the variable before it is initialized. (This can be a source of bugs and is one of the important misfeatures that `let` corrects: if you declare a variable with `let` but attempt to use it before the `let` statement runs, you will get an actual error instead of just seeing an `undefined` value.)

Using Undeclared Variables

In strict mode (§5.6.3), if you attempt to use an undeclared variable, you'll get a reference error when you run your code. Outside of strict mode, however, if you assign a value to a name that has not been declared with `let`, `const`, or `var`, you'll end up creating a new global variable. It will be a global no matter now deeply nested within functions and blocks your code is, which is almost certainly not what you want, is bug-prone, and is one of the best reasons for using strict mode!

Global variables created in this accidental way are like global variables declared with `var`: they define properties of the global object. But unlike the properties defined by proper `var` declarations, these properties *can* be deleted with the `delete` operator (§4.13.4).

3.10.3 Destructuring Assignment

ES6 implements a kind of compound declaration and assignment syntax known as *destructuring assignment*. In a destructuring assignment, the value on the righthand side of the equals sign is an array or object (a "structured" value), and the lefthand 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. Destructuring assignment is perhaps most commonly used to initialize variables as part of a `const`, `let`, or `var` declaration statement, but it can also be done in regular assignment expressions (with variables that have already been declared). And, as we'll see in §8.3.5, destructuring can also be used when defining the parameters to a function.

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
[x,y]               // => [3,2]: the incremented and swapped values
```

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 toPolar(x, y) {
    return [Math.sqrt(x*x+y*y), Math.atan2(y,x)];
}

// Convert polar to Cartesian coordinates
function toCartesian(r, theta) {
    return [r*Math.cos(theta), r*Math.sin(theta)];
}
```

```
let [r,theta] = toPolar(1.0, 1.0);  // r == Math.sqrt(2); theta == Math.PI/4
let [x,y] = toCartesian(r,theta);   // [x, y] == [1.0, 1,0]
```

We saw that variables and constants can be declared as part of JavaScript's various `for` loops. It is possible to use variable destructuring in this context as well. Here is a code that loops over the name/value pairs of all properties of an object and uses destructuring assignment to convert those pairs from two-element arrays into individual variables:

```
let o = { x: 1, y: 2 }; // The object we'll loop over
for(const [name, value] of Object.entries(o)) {
    console.log(name, value); // Prints "x 1" and "y 2"
}
```

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
```

If you want to collect all unused or remaining values into a single variable when destructuring an array, use three dots (`...`) before the last variable name on the lefthand side:

```
let [x, ...y] = [1,2,3,4];  // y == [2,3,4]
```

We'll see three dots used this way again in §8.3.2, where they are used to indicate that all remaining function arguments should be collected into a single array.

Destructuring assignment can be used with nested arrays. In this case, the lefthand side of the assignment should look like a nested array literal:

```
let [a, [b, c]] = [1, [2,2.5], 3]; // a == 1; b == 2; c == 2.5
```

A powerful feature of array destructuring is that it does not actually require an array! You can use any *iterable* object (Chapter 12) on the righthand side of the assignment; any object that can be used with a `for/of` loop (§5.4.4) can also be destructured:

```
let [first, ...rest] = "Hello"; // first == "H"; rest == ["e","l","l","o"]
```

Destructuring assignment can also be performed when the righthand side is an object value. In this case, the lefthand side of the assignment looks something like an object literal: a comma-separated list of variable names within curly braces:

```
let transparent = {r: 0.0, g: 0.0, b: 0.0, a: 1.0}; // A RGBA color
let {r, g, b} = transparent;  // r == 0.0; g == 0.0; b == 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 const sin=Math.sin, cos=Math.cos, tan=Math.tan
const {sin, cos, tan} = Math;
```

Notice in the code here that the `Math` object has many properties other than the three that are destructured into individual variables. Those that are not named are simply ignored. If the lefthand side of this assignment had included a variable whose name was not a property of `Math`, that variable would simply be assigned `undefined`.

In each of these object destructuring examples, we have chosen variable names that match the property names of the object we're destructuring. This keeps the syntax simple and easy to understand, but it is not required. Each of the identifiers on the lefthand side of an object destructuring assignment can also be a colon-separated pair of identifiers, where the first is the name of the property whose value is to be assigned and the second is the name of the variable to assign it to:

```
// Same as const cosine = Math.cos, tangent = Math.tan;
const { cos: cosine, tan: tangent } = Math;
```

I find that object destructuring syntax becomes too complicated to be useful when the variable names and property names are not the same, and I tend to avoid the shorthand in this case. If you choose to use it, remember that property names are always on the left of the colon, in both object literals and on the left of an object destructuring assignment.

Destructuring assignment becomes even more complicated when it is used with nested objects, or arrays of objects, or objects of arrays, but it is legal:

```
let points = [{x: 1, y: 2}, {x: 3, y: 4}];     // An array of two point objects
let [{x: x1, y: y1}, {x: x2, y: y2}] = points; // destructured into 4 variables.
(x1 === 1 && y1 === 2 && x2 === 3 && y2 === 4) // => true
```

Or, instead of destructuring an array of objects, we could destructure an object of arrays:

```
let points = { p1: [1,2], p2: [3,4] };         // An object with 2 array props
let { p1: [x1, y1], p2: [x2, y2] } = points;   // destructured into 4 vars
(x1 === 1 && y1 === 2 && x2 === 3 && y2 === 4) // => true
```

Complex destructuring syntax like this can be hard to write and hard to read, and you may be better off just writing out your assignments explicitly with traditional code like `let x1 = points.p1[0];`.

Understanding Complex Destructuring

If you find yourself working with code that uses complex destructuring assignments, there is a useful regularity that can help you make sense of the complex cases. Think first about a regular (single-value) assignment. After the assignment is done, you can take the variable name from the lefthand side of the assignment and use it as an expression in your code, where it will evaluate to whatever value you assigned it. The same thing is true for destructuring assignment. The lefthand side of a destructuring assignment looks like an array literal or an object literal (§6.2.1 and §6.10). After the assignment has been done, the lefthand side will actually work as a valid array literal or object literal elsewhere in your code. You can check that you've written a destructuring assignment correctly by trying to use the lefthand side on the righthand side of another assignment expression:

```
// Start with a data structure and a complex destructuring
let points = [{x: 1, y: 2}, {x: 3, y: 4}];
let [{x: x1, y: y1}, {x: x2, y: y2}] = points;

// Check your destructuring syntax by flipping the assignment around
let points2 = [{x: x1, y: y1}, {x: x2, y: y2}]; // points2 == points
```

3.11 Summary

Some key points to remember about this chapter:

- How to write and manipulate numbers and strings of text in JavaScript.
- How to work with JavaScript's other primitive types: booleans, Symbols, `null`, and `undefined`.
- The differences between immutable primitive types and mutable reference types.
- How JavaScript converts values implicitly from one type to another and how you can do so explicitly in your programs.
- How to declare and initialize constants and variables (including with destructuring assignment) and the lexical scope of the variables and constants you declare.

CHAPTER 4

Expressions and Operators

This chapter documents JavaScript expressions and the operators with which many of those expressions are built. An *expression* is a phrase of JavaScript that can be *evaluated* 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 §3.2. String literals were documented in §3.3. The regular expression literal syntax was introduced in §3.3.5 and will be documented in detail in §11.3.

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 §3.4 and §3.5. 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 a reference to a variable, constant, or property of the global object:

```
i             // Evaluates to the value of the variable i.
sum           // Evaluates to the value of the variable sum.
undefined     // The value of the "undefined" property of the global object
```

When any identifier appears by itself in a program, JavaScript assumes it is a variable or constant or property of the global object and looks up its value. If no variable with that name exists, 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:

```
let 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:

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

A single trailing comma is allowed after the last expression in an array initializer and does not create an undefined element. However, any array access expression for an index after that of the last expression will necessarily evaluate to undefined.

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:

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

In ES6, object literals have a much more feature-rich syntax (you can find details in §6.10). Object literals can be nested. For example:

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

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.
let 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. And in ES6 and later, function expressions can use a compact new "arrow function" syntax. 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:

```
let o = {x: 1, y: {z: 3}}; // An example object
let 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 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 the name when you write the program. If the property

name 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 §6.3.1 for an example).

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

4.4.1 Conditional Property Access

ES2020 adds two new kinds of property access expressions:

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

In JavaScript, the values `null` and `undefined` are the only two values that do not have properties. In a regular property access expression using `.` or `[]`, you get a TypeError if the expression on the left evaluates to `null` or `undefined`. You can use `?.` and `?.[]` syntax to guard against errors of this type.

Consider the expression `a?.b`. If `a` is `null` or `undefined`, then the expression evaluates to `undefined` without any attempt to access the property `b`. If `a` is some other value, then `a?.b` evaluates to whatever `a.b` would evaluate to (and if `a` does not have a property named `b`, then the value will again be `undefined`).

This form of property access expression is sometimes called "optional chaining" because it also works for longer "chained" property access expressions like this one:

```
let a = { b: null };
a.b?.c.d   // => undefined
```

`a` is an object, so `a.b` is a valid property access expression. But the value of `a.b` is `null`, so `a.b.c` would throw a TypeError. By using `?.` instead of `.` we avoid the TypeError, and `a.b?.c` evaluates to `undefined`. This means that `(a.b?.c).d` will throw a TypeError, because that expression attempts to access a property of the value `undefined`. But—and this is a very important part of "optional chaining"—`a.b?.c.d` (without the parentheses) simply evaluates to `undefined` and does not throw an error. This is because property access with `?.` is "short-circuiting": if the subexpression to the left of `?.` evaluates to `null` or `undefined`, then the entire expression immediately evaluates to `undefined` without any further property access attempts.

Of course, if `a.b` is an object, and if that object has no property named `c`, then `a.b?.c.d` will again throw a TypeError, and we will want to use another conditional property access:

```
let a = { b: {} };
a.b?.c?.d  // => undefined
```

Conditional property access is also possible using `?.[]` instead of `[]`. In the expression `a?.[b][c]`, if the value of `a` is `null` or `undefined`, then the entire expression immediately evaluates to `undefined`, and subexpressions `b` and `c` are never even evaluated. If either of those expressions has side effects, the side effect will not occur if `a` is not defined:

```
let a;          // Oops, we forgot to initialize this variable!
let index = 0;
try {
    a[index++]; // Throws TypeError
} catch(e) {
    index       // => 1: increment occurs before TypeError is thrown
}
a?.[index++]    // => undefined: because a is undefined
index           // => 1: not incremented because ?.[] short-circuits
a[index++]      // !TypeError: can't index undefined.
```

Conditional property access with `?.` and `?.[]` is one of the newest features of JavaScript. As of early 2020, this new syntax is supported in the current or beta versions of most major browsers.

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 function, a TypeError is thrown. 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` keyword while

the body of the function is being executed. This enables an object-oriented programming paradigm in which functions (which we call "methods" when used this way) operate on the object of which they are part. See Chapter 9 for details.

4.5.1 Conditional Invocation

In ES2020, you can also invoke a function using `?.()` instead of `()`. Normally when you invoke a function, if the expression to the left of the parentheses is `null` or `undefined` or any other non-function, a TypeError is thrown. With the new `?.()` invocation syntax, if the expression to the left of the `?.` evaluates to `null` or `undefined`, then the entire invocation expression evaluates to `undefined` and no exception is thrown.

Array objects have a `sort()` method that can optionally be passed a function argument that defines the desired sorting order for the array elements. Before ES2020, if you wanted to write a method like `sort()` that takes an optional function argument, you would typically use an `if` statement to check that the function argument was defined before invoking it in the body of the `if`:

```
function square(x, log) { // The second argument is an optional function
    if (log) {            // If the optional function is passed
        log(x);           // Invoke it
    }
    return x * x;         // Return the square of the argument
}
```

With this conditional invocation syntax of ES2020, however, you can simply write the function invocation using `?.()`, knowing that invocation will only happen if there is actually a value to be invoked:

```
function square(x, log) { // The second argument is an optional function
    log?.(x);             // Call the function if there is one
    return x * x;         // Return the square of the argument
}
```

Note, however, that `?.()` only checks whether the lefthand side is `null` or `undefined`. It does not verify that the value is actually a function. So the `square()` function in this example would still throw an exception if you passed two numbers to it, for example.

Like conditional property access expressions (§4.4.1), function invocation with `?.()` is short-circuiting: if the value to the left of `?.` is `null` or `undefined`, then none of the argument expressions within the parentheses are evaluated:

```
let f = null, x = 0;
try {
    f(x++); // Throws TypeError because f is null
} catch(e) {
    x       // => 1: x gets incremented before the exception is thrown
}
```

```
f?.(x++)    // => undefined: f is null, but no exception thrown
x           // => 1: increment is skipped because of short-circuiting
```

Conditional invocation expressions with `?.()` work just as well for methods as they do for functions. But because method invocation also involves property access, it is worth taking a moment to be sure you understand the differences between the following expressions:

```
o.m()     // Regular property access, regular invocation
o?.m()    // Conditional property access, regular invocation
o.m?.()   // Regular property access, conditional invocation
```

In the first expression, `o` must be an object with a property `m` and the value of that property must be a function. In the second expression, if `o` is `null` or `undefined`, then the expression evaluates to `undefined`. But if `o` has any other value, then it must have a property `m` whose value is a function. And in the third expression, `o` must not be `null` or `undefined`. If it does not have a property `m`, or if the value of that property is `null`, then the entire expression evaluates to `undefined`.

Conditional invocation with `?.()` is one of the newest features of JavaScript. As of the first months of 2020, this new syntax is supported in the current or beta versions of most major browsers.

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
```

The value of an object creation expression is the newly created object. 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	any→num
~	Invert bits	R	1	int→int
!	Invert boolean value	R	1	bool→bool
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
**	Exponentiate	R	2	num,num→num
*, /, %	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 alphabetical order	L	2	str,str→bool
instanceof	Test object class	L	2	obj,func→bool
in	Test whether property exists	L	2	any,obj→bool
==	Test for non-strict equality	L	2	any,any→bool
!=	Test for non-strict inequality	L	2	any,any→bool
===	Test for strict equality	L	2	any,any→bool

Operator	Operation	A	N	Types
`!==`	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 1st defined operand	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 2nd	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 §3.9) 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.

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.

4.7.3 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.

4.7.4 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:

```
// my is an object with a property named functions whose value is an
// array of functions. We invoke function number x, passing it argument
// y, and then we ask for the type of the value returned.
typeof my.functions[x](y)
```

Although `typeof` is one of the highest-priority operators, the `typeof` operation is performed on the result of the property access, array index, and function invocation, all of which have higher priority than operators.

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.

When new operators are added to JavaScript, they do not always fit naturally into this precedence scheme. The `??` operator (§4.13.2) is shown in the table as lower-precedence than `||` and `&&`, but, in fact, its precedence relative to those operators is not defined, and ES2020 requires you to explicitly use parentheses if you mix `??` with either `||` or `&&`. Similarly, the new `**` exponentiation operator does not have a well-defined precedence relative to the unary negation operator, and you must use parentheses when combining negation with exponentiation.

4.7.5 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:

```
y = a ** b ** c;
x = ~-y;
```

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

are equivalent to:

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

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

4.7.6 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 exponentiation, 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.

Most of these arithmetic operators (except as noted as follows) can be used with BigInt (see §3.2.5) operands or with regular numbers, as long as you don't mix the two types.

The basic arithmetic operators are `**` (exponentiation), `*` (multiplication), `/` (division), `%` (modulo: remainder after division), `+` (addition), and `-` (subtraction). As noted, we'll discuss the `+` operator in a section of its own. The other five basic operators simply evaluate their operands, convert the values to numbers if necessary, and then compute the power, product, quotient, remainder, or difference. 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 (almost always) NaN.

The ** operator has higher precedence than *, /, and % (which in turn have higher precedence than + and -). Unlike the other operators, ** works right-to-left, so 2**2**3 is the same as 2**8, not 4**3. There is a natural ambiguity to expressions like -3**2. Depending on the relative precedence of unary minus and exponentiation, that expression could mean (-3)**2 or -(3**2). Different languages handle this differently, and rather than pick sides, JavaScript simply makes it a syntax error to omit parentheses in this case, forcing you to write an unambiguous expression. ** is JavaScript's newest arithmetic operator: it was added to the language with ES2016. The Math.pow() function has been available since the earliest versions of JavaScript, however, and it performs exactly the same operation as the ** operator.

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 conversion 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 §3.9.3. Date objects are converted by their `toString()` method, and all other objects are converted via `valueOf()`, if that method returns a primitive value. However, most objects do not have a useful `valueOf()` method, 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. This operator may not be used with BigInt values, since they cannot be converted to regular numbers.

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:

```
let i = 1, j = ++i;    // i and j are both 2
let n = 1, m = n++;    // n is 2, m 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 (§5.4.3).

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. 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. These operators are not commonly used in JavaScript programming, and if you are not familiar with the binary representation of integers, including the two's complement representation of negative integers, you can probably skip this section.

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.

All of these bitwise operators except `>>>` can be used with regular number operands or with BigInt (see §3.2.5) operands.

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 positive 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. `7 >> 1` evaluates to 3, for example, but note that 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. This is useful when you want to treat signed 32-bit values as if they are unsigned integers. `-1 >> 4` evaluates to −1, but `-1 >>> 4` evaluates to `0x0FFFFFFF`, for example. This is the only one of the JavaScript bitwise operators that cannot be used with BigInt values. BigInt does not represent negative numbers by setting the high bit the way that 32-bit integers do, and this operator only makes sense for that particular two's complement representation.

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.

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."

The =, ==, and === operators

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 as "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 == operator is a legacy feature of JavaScript and is widely considered to be a source of bugs. You should almost always use === instead of ==, and `!==` instead of `!=`.

As mentioned in §3.8, 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. Similarly, two arrays that have the same elements in the same order are not equal to each other.

Strict equality

The strict equality operator === evaluates its operands, 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. (This is surprising, but the `NaN` value is never equal to any other value, including itself! To check whether a value `x` is `NaN`, use `x !== x`, or the global `isNaN()` function.)
- 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 §3.3) 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 is not considered equal to the `===` or `==` operators.
- 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.

Equality with type conversion

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 previously. 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. It uses 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 §3.9.3 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.

— Any other combinations of values are not equal.

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

```
"1" == true  // => 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 alphabetical) 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 §3.9.3; 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`. Although the arithmetic operators do not allow BigInt values to be mixed with regular numbers, the comparison operators do allow comparisons between numbers and BigInts.

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, try the `String.localeCompare()` method, which also takes locale-specific definitions of alphabetical order into account. For case-insensitive comparisons, you can convert the strings to all lowercase or all uppercase using `String.toLowerCase()` or `String.toUpperCase()`. And, for a more general and better localized string comparison tool, use the Intl.Collator class described in §11.7.3.

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         // => 3: addition.
"1" + "2"     // => "12": concatenation.
"1" + 2       // => "12": 2 is converted to "2".
11 < 3        // => false: numeric comparison.
"11" < "3"    // => true: string comparison.
"11" < 3      // => false: numeric comparison, "11" converted to 11.
"one" < 3     // => false: numeric comparison, "one" converted to NaN.
```

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 a string, symbol, or value that 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:

```
let 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

let data = [7,8,9];        // An array with elements (indices) 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:

```
let d = new Date();  // Create a new object with the Date() constructor
d instanceof Date    // => true: d was created with Date()
d instanceof Object  // => true: all objects are instances of Object
d instanceof Number  // => false: d is not a Number object
let a = [1, 2, 3];   // Create an array with array literal syntax
a instanceof Array   // => true: a is an array
a instanceof Object  // => true: all arrays are objects
a instanceof RegExp  // => 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 righthand side is not a class of objects, 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 §6.3.2. 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 subclass 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 §3.4.

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 §3.4 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 this description 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 righthand 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:

```
let o = {x: 1};
let 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 this code example, 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 righthand side of `&&`. Whether those side effects occur depends on the value of the lefthand 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 short-circuits and returns that truthy value without ever evaluating the expression on the right. If, on the other hand, the value of the first operand is falsy, then `||` evaluates its second operand 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 maxWidth is truthy, use that. Otherwise, look for a value in
// the preferences object. If that is not truthy, use a hardcoded constant.
let max = maxWidth || preferences.maxWidth || 500;
```

Note that if 0 is a legal value for `maxWidth`, then this code will not work correctly, since 0 is a falsy value. See the `??` operator (§4.13.2) for an alternative.

Prior to ES6, this idiom is often used in functions 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
}
```

In ES6 and later, however, this trick is no longer needed because the default parameter value could simply be written in the function definition itself: `function copy(o, p={}) { ... }`.

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 §3.9.2).

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 laws of Boolean algebra here that we can express using JavaScript syntax:

```
// DeMorgan's Laws
!(p && q) === (!p || !q)  // => true: for all values of p and q
!(p || q) === (!p && !q)  // => true: for all values of p and 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 += salesTax;
```

is equivalent to this one:

```
total = total + salesTax;
```

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

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
^=	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. In particular, `eval()` can be a security hole, and you should never pass any string derived from user input to `eval()`. With a language as complicated as JavaScript, there is no way to sanitize user input to make it safe to use with `eval()`. Because of these security issues, some web servers use the HTTP "Content-Security-Policy" header to disable `eval()` for an entire website.

The subsections that follow explain the basic use of `eval()` and 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. 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:

```
let f = eval;
let g = f;
```

If this is allowed, then the interpreter can't know for sure which functions call `eval()`, so it cannot optimize aggressively. This issue could have been avoided if `eval()` was an operator (and a reserved word). We'll learn (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 declares a new local variable `y`. On the other hand, if the evaluated string uses `let` or `const`, the variable or constant declared will be local to the evaluation and will not be defined in the calling environment.

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? The JavaScript specification declares that when `eval()` is invoked by any name other than "eval", it should evaluate the string as if it were top-level global code. The evaluated code may define new global variables or global functions, and it may set global variables, but it will not use or modify any variables local to the calling function, and will not, therefore, interfere with local optimizations.

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. (Both direct and indirect calls can define new variables only with `var`. Uses of `let` and `const` inside an evaluated string create variables and constants that are local to the evaluation and do not alter the calling or global environment.)

The following code demonstrates:

```
const geval = eval;               // Using another name does a global eval
let x = "global", y = "global";   // Two global variables
function f() {                    // This function does a local eval
    let 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
    let 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 that 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.

4.12.3 Strict eval()

Strict mode (see §5.6.3) 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 (§5.3.1), 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 First-Defined (??)

The first-defined operator `??` evaluates to its first defined operand: if its left operand is not `null` and not `undefined`, it returns that value. Otherwise, it returns the value of the right operand. Like the `&&` and `||` operators, `??` is short-circuiting: it only evaluates its second operand if the first operand evaluates to `null` or `undefined`. If the expression `a` has no side effects, then the expression `a ?? b` is equivalent to:

```
(a !== null && a !== undefined) ? a : b
```

`??` is a useful alternative to `||` (§4.10.2) when you want to select the first *defined* operand rather than the first truthy operand. Although `||` is nominally a logical OR operator, it is also used idiomatically to select the first non-falsy operand with code like this:

```
// If maxWidth is truthy, use that. Otherwise, look for a value in
// the preferences object. If that is not truthy, use a hardcoded constant.
let max = maxWidth || preferences.maxWidth || 500;
```

The problem with this idiomatic use is that zero, the empty string, and `false` are all falsy values that may be perfectly valid in some circumstances. In this code example, if `maxWidth` is zero, that value will be ignored. But if we change the `||` operator to `??`, we end up with an expression where zero is a valid value:

```
// If maxWidth is defined, use that. Otherwise, look for a value in
// the preferences object. If that is not defined, use a hardcoded constant.
let max = maxWidth ?? preferences.maxWidth ?? 500;
```

Here are more examples showing how `??` works when the first operand is falsy. If that operand is falsy but defined, then `??` returns it. It is only when the first operand is "nullish" (i.e., `null` or `undefined`) that this operator evaluates and returns the second operand:

```
let options = { timeout: 0, title: "", verbose: false, n: null };
options.timeout ?? 1000     // => 0: as defined in the object
options.title ?? "Untitled" // => "": as defined in the object
options.verbose ?? true     // => false: as defined in the object
options.quiet ?? false      // => false: property is not defined
options.n ?? 10             // => 10: property is null
```

Note that the `timeout`, `title`, and `verbose` expressions here would have different values if we used `||` instead of `??`.

The ?? operator is similar to the && and || operators but does not have higher precedence or lower precedence than they do. If you use it in an expression with either of those operators, you must use explicit parentheses to specify which operation you want to perform first:

```
(a ?? b) || c   // ?? first, then ||
a ?? (b || c)   // || first, then ??
a ?? b || c     // SyntaxError: parentheses are required
```

The ?? operator is defined by ES2020, and as of early 2020, is newly supported by current or beta versions of all major browsers. This operator is formally called the "nullish coalescing" operator, but I avoid that term because this operator selects one of its operands but does not "coalesce" them in any way that I can see.

4.13.3 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. Table 4-3 specifies the value of the typeof operator for any JavaScript value.

Table 4-3. Values returned by the typeof operator

x	typeof x
undefined	"undefined"
null	"object"
true or false	"boolean"
any number or NaN	"number"
any BigInt	"bigint"
any string	"string"
any symbol	"symbol"
any function	"function"
any nonfunction object	"object"

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

```
// If the value is a string, wrap it in quotes, otherwise, convert
(typeof value === "string") ? "'" + value + "'" : value.toString()
```

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.

Although JavaScript functions are a kind of object, the typeof operator considers functions to be sufficiently different that they have their own return value.

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 §14.4.3), or the `constructor` property (see §9.2.2 and §14.3).

4.13.4 The delete Operator

`delete` is a unary operator that attempts to delete the object property or array element specified as its operand. 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:

```
let 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

let 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.length               // => 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* (§7.3).

`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: non-configurable properties (§14.1) are immune from deletion.

In 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 non-configurable (i.e., nondeleteable) property. 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:

```
let o = {x: 1, y: 2};
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 1;     // This makes no sense, but it just returns true.
// Can't delete a variable; returns false, or SyntaxError in strict mode.
delete o;
```

```
// Undeletable property: returns false, or TypeError in strict mode.
delete Object.prototype;
```

We'll see the `delete` operator again in §6.4.

4.13.5 The await Operator

`await` was introduced in ES2017 as a way to make asynchronous programming more natural in JavaScript. You will need to read Chapter 13 to understand this operator. Briefly, however, `await` expects a Promise object (representing an asynchronous computation) as its sole operand, and it makes your program behave as if it were waiting for the asynchronous computation to complete (but it does this without actually blocking, and it does not prevent other asynchronous operations from proceeding at the same time). The value of the `await` operator is the fulfillment value of the Promise object. Importantly, `await` is only legal within functions that have been declared asynchronous with the `async` keyword. Again, see Chapter 13 for full details.

4.13.6 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 `void` operator is so obscure that it is difficult to come up with a practical example of its use. One case would be when you want to define a function that returns nothing but also uses the arrow function shortcut syntax (see §8.1.3) where the body of the function is a single expression that is evaluated and returned. If you are evaluating the expression solely for its side effects and do not want to return its value, then the simplest thing is to use curly braces around the function body. But, as an alternative, you could also use the void operator in this case:

```
let counter = 0;
const increment = () => void counter++;
increment()   // => undefined
counter       // => 1
```

4.13.7 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 lefthand expression is always evaluated, but its value is discarded, which means that it only makes sense to use the comma operator when the lefthand expression has side effects. The only situation in which the comma operator is commonly used is with a `for` loop (§5.4.3) that has multiple loop variables:

```
// The first comma below is part of the syntax of the let 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(let i=0,j=10; i < j; i++,j--) {
    console.log(i+j);
}
```

4.14 Summary

This chapter covers a wide variety of topics, and there is lots of reference material here that you may want to reread in the future as you continue to learn JavaScript. Some key points to remember, however, are these:

- Expressions are the phrases of a JavaScript program.
- Any expression can be *evaluated* to a JavaScript value.
- Expressions can also have side effects (such as variable assignment) in addition to producing a value.
- Simple expressions such as literals, variable references, and property accesses can be combined with operators to produce larger expressions.
- JavaScript defines operators for arithmetic, comparisons, Boolean logic, assignment, and bit manipulation, along with some miscellaneous operators, including the ternary conditional operator.
- The JavaScript `+` operator is used to both add numbers and concatenate strings.
- The logical operators `&&` and `||` have special "short-circuiting" behavior and sometimes only evaluate one of their arguments. Common JavaScript idioms require you to understand the special behavior of these operators.

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 one another with periods, JavaScript statements are terminated with semicolons (§2.6). 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 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
: Statements like `if` and `switch` that make the JavaScript interpreter execute or skip other statements depending on the value of an expression

Loops
: Statements like `while` and `for` that execute other statements repetitively

Jumps
: 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. 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:

```
console.log(debugMessage);
displaySpinner(); // A hypothetical function to display a spinner in a web app.
```

These function calls are expressions, but they have side effects that affect the host environment or program state, and they 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 (§4.13.7) 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.

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 §5.4.3):

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

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

5.3.1 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 §3.4 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");
}
```

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, and I advise you to adopt this practice. In this printed book, I place a premium on keeping example code vertically compact, and I do not always follow my own advice on this matter.

5.3.2 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
        }
    }
}
```

5.3.3 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. 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 this code. 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% 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 shown, the `default:` label appears at the end of the `switch` body,

1 The fact that the `case` expressions are evaluated at runtime 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*.

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.

5.4 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 five looping statements: `while`, `do/while`, `for`, `for/of` (and its `for/await` variant), and `for/in`. The following subsections 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.

5.4.1 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:

```
let 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.

5.4.2 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) {
    let 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.

5.4.3 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 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, it is either 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(let 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:

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

2 When we consider the `continue` statement in §5.5.3, we'll see that this `while` loop is not an exact equivalent of the `for` loop.

```
        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 this code 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)`.

5.4.4 for/of

ES6 defines a new loop statement: `for/of`. This new kind of loop uses the `for` keyword but is a completely different kind of loop than the regular `for` loop. (It is also completely different than the older `for/in` loop that we'll describe in §5.4.5.)

The `for/of` loop works with *iterable* objects. We'll explain exactly what it means for an object to be iterable in Chapter 12, but for this chapter, it is enough to know that arrays, strings, sets, and maps are iterable: they represent a sequence or set of elements that you can loop or iterate through using a `for/of` loop.

Here, for example, is how we can use `for/of` to loop through the elements of an array of numbers and compute their sum:

```
let data = [1, 2, 3, 4, 5, 6, 7, 8, 9], sum = 0;
for(let element of data) {
    sum += element;
}
sum       // => 45
```

Superficially, the syntax looks like a regular `for` loop: the `for` keyword is followed by parentheses that contain details about what the loop should do. In this case, the parentheses contain a variable declaration (or, for variables that have already been declared, simply the name of the variable) followed by the `of` keyword and an expression that evaluates to an iterable object, like the `data` array in this case. As with all loops, the body of a `for/of` loop follows the parentheses, typically within curly braces.

In the code just shown, the loop body runs once for each element of the `data` array. Before each execution of the loop body, the next element of the array is assigned to the element variable. Array elements are iterated in order from first to last.

Arrays are iterated "live"—changes made during the iteration may affect the outcome of the iteration. If we modify the preceding code by adding the line `data.push(sum);` inside the loop body, then we create an infinite loop because the iteration can never reach the last element of the array.

for/of with objects

Objects are not (by default) iterable. Attempting to use `for/of` on a regular object throws a TypeError at runtime:

```
let o = { x: 1, y: 2, z: 3 };
for(let element of o) { // Throws TypeError because o is not iterable
    console.log(element);
}
```

If you want to iterate through the properties of an object, you can use the `for/in` loop (introduced in §5.4.5), or use `for/of` with the `Object.keys()` method:

```
let o = { x: 1, y: 2, z: 3 };
let keys = "";
for(let k of Object.keys(o)) {
    keys += k;
}
keys  // => "xyz"
```

This works because `Object.keys()` returns an array of property names for an object, and arrays are iterable with `for/of`. Note also that this iteration of the keys of an object is not live as the array example above was—changes to the object `o` made in the loop body will have no effect on the iteration. If you don't care about the keys of an object, you can also iterate through their corresponding values like this:

```
let sum = 0;
for(let v of Object.values(o)) {
    sum += v;
}
sum // => 6
```

And if you are interested in both the keys and the values of an object's properties, you can use `for/of` with `Object.entries()` and destructuring assignment:

```
let pairs = "";
for(let [k, v] of Object.entries(o)) {
    pairs += k + v;
}
pairs  // => "x1y2z3"
```

`Object.entries()` returns an array of arrays, where each inner array represents a key/value pair for one property of the object. We use destructuring assignment in this code example to unpack those inner arrays into two individual variables.

for/of with strings

Strings are iterable character-by-character in ES6:

```
let frequency = {};
for(let letter of "mississippi") {
    if (frequency[letter]) {
        frequency[letter]++;
    } else {
        frequency[letter] = 1;
    }
}
frequency   // => {m: 1, i: 4, s: 4, p: 2}
```

Note that strings are iterated by Unicode codepoint, not by UTF-16 character. The string "I ❤ 🦙" has a `.length` of 5 (because the two emoji characters each require two UTF-16 characters to represent). But if you iterate that string with `for/of`, the loop body will run three times, once for each of the three code points "I", "❤", and "🦙".

for/of with Set and Map

The built-in ES6 Set and Map classes are iterable. When you iterate a Set with `for/of`, the loop body runs once for each element of the set. You could use code like this to print the unique words in a string of text:

```
let text = "Na na na na na na na na Batman!";
let wordSet = new Set(text.split(" "));
let unique = [];
for(let word of wordSet) {
    unique.push(word);
}
unique // => ["Na", "na", "Batman!"]
```

Maps are an interesting case because the iterator for a Map object does not iterate the Map keys, or the Map values, but key/value pairs. Each time through the iteration, the iterator returns an array whose first element is a key and whose second element is the corresponding value. Given a Map `m`, you could iterate and destructure its key/value pairs like this:

```
let m = new Map([[1, "one"]]);
for(let [key, value] of m) {
    key    // => 1
    value  // => "one"
}
```

Asynchronous iteration with for/await

ES2018 introduces a new kind of iterator, known as an *asynchronous iterator*, and a variant on the `for/of` loop, known as the `for/await` loop that works with asynchronous iterators.

You'll need to read Chapters 12 and 13 in order to understand the `for/await` loop, but here is how it looks in code:

```
// Read chunks from an asynchronously iterable stream and print them out
async function printStream(stream) {
    for await (let chunk of stream) {
        console.log(chunk);
    }
}
```

5.4.5 for/in

A `for/in` loop looks a lot like a `for/of` loop, with the `of` keyword changed to `in`. While a `for/of` loop requires an iterable object after the `of`, a `for/in` loop works with any object after the `in`. The `for/of` loop is new in ES6, but `for/in` has been part of JavaScript since the very beginning (which is why it has the more natural sounding syntax).

The `for/in` statement loops through the property names of a specified object. The syntax looks like this:

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

variable typically names a variable, but it may be a variable declaration or anything suitable as the left-hand 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.

And you might use a `for/in` loop like this:

```
for(let 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. 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:

```
let o = { x: 1, y: 2, z: 3 };
let 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 previous code with this line enumerates the array indexes 0, 1, and 2:

```
for(let i in a) console.log(i);
```

I find that a common source of bugs in my own code is the accidental use of `for/in` with arrays when I meant to use `for/of`. When working with arrays, you almost always want to use `for/of` instead of `for/in`.

The `for/in` loop does not actually enumerate all properties of an object. It does not enumerate properties whose names are symbols. And of the properties whose names are strings, it only loops over the *enumerable* properties (see §14.1). The various built-in methods defined by core JavaScript are not enumerable. All objects have a `toString()` method, for example, but the `for/in` loop does not enumerate this `toString` property. In addition to built-in methods, many other properties of the built-in objects are non-enumerable. All properties and methods defined by your code are enumerable, by default. (You can make them non-enumerable using techniques explained in §14.1.)

Enumerable inherited properties (see §6.3.2) are also enumerated by the `for/in` loop. This means that if you use `for/in` loops and also use code that defines properties that are inherited by all objects, then your loop may not behave in the way you expect. For this reason, many programmers prefer to use a `for/of` loop with `Object.keys()` instead of a `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 may or may not be enumerated. See §6.6.1 for more information on the order in which `for/in` enumerates the properties of an object.

5.5 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 `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 is a kind of interim return from a generator function. 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 about each of these jump statements are in the sections that follow.

5.5.1 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 in the following subsections. 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.

5.5.2 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(let 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 §2.6.)

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:

```
let matrix = getData();  // Get a 2D array of numbers from somewhere
// Now sum all the numbers in the matrix.
let sum = 0, success = false;
// Start with a labeled statement that we can break out of if errors occur
```

```
computeSum: if (matrix) {
    for(let x = 0; x < matrix.length; x++) {
        let row = matrix[x];
        if (!row) break computeSum;
        for(let y = 0; y < row.length; y++) {
            let cell = row[y];
            if (isNaN(cell)) break computeSum;
            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.

5.5.3 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/of` or `for/in` loop, the loop starts over with the next iterated value or 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(let 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, as with the `break` statement, line breaks are not allowed between the `continue` statement and its *labelname*.

5.5.4 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)                          // => 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 displayObject(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 (§2.6), you cannot include a line break between the `return` keyword and the expression that follows it.

5.5.5 yield

The `yield` statement is much like the `return` statement but is used only in ES6 generator functions (see §12.3) to produce the next value in the generated sequence of values without actually returning:

```
// A generator function that yields a range of integers
function* range(from, to) {
    for(let i = from; i <= to; i++) {
        yield i;
    }
}
```

In order to understand `yield`, you must understand iterators and generators, which will not be covered until Chapter 12. `yield` is included here for completeness, however. (Technically, though, `yield` is an operator rather than a statement, as explained in §12.4.2.)

5.5.6 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. 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
    let f;
    for(f = 1; x > 1; f *= x, x--) /* empty */ ;
    return f;
}
factorial(4)   // => 24
```

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.

5.5.7 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 generally 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. 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
    let n = Number(prompt("Please enter a positive integer", ""));
    // Compute the factorial of the number, assuming the input is valid
    let 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`.

Bare Catch Clauses

Occasionally you may find yourself using a `catch` clause solely to detect and stop the propagation of an exception, even though you do not care about the type or the value of the exception. In ES2019 and later, you can omit the parentheses and the identifier and use the `catch` keyword bare in this case. Here is an example:

```
// Like JSON.parse(), but return undefined instead of throwing an error
function parseJSON(s) {
    try {
        return JSON.parse(s);
    } catch {
        // Something went wrong but we don't care what it was
        return undefined;
    }
}
```

5.6 Miscellaneous Statements

This section describes the remaining three JavaScript statements—`with`, `debugger`, and `"use strict"`.

5.6.1 with

The `with` statement runs a block of code as if the properties of a specified object were variables in scope for that code. It has the following syntax:

```
with (object)
    statement
```

This statement creates a temporary scope with the properties of *object* as variables and then executes *statement* within that scope.

The `with` statement is forbidden in strict mode (see §5.6.3) 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 significantly 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 treat the properties of the form object like variables:

```
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]`. It is just as simple, of course, to avoid the `with` statement and write the preceding code like this:

```
let f = document.forms[0];
f.name.value = "";
f.address.value = "";
f.email.value = "";
```

Note that if you use `const` or `let` or `var` to declare a variable or constant within the body of a `with` statement, it creates an ordinary variable and does not define a new property within the specified object.

5.6.2 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.

Note that it is not enough to have a debugger available: the `debugger` statement won't start the debugger for you. If you're using a web browser and have the developer tools console open, however, this statement will cause a breakpoint.

5.6.3 "use strict"

`"use strict"` is a *directive* introduced in ES5. 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).
- It can appear only at the start of a script or at the start of a function body, before any real statements have appeared.

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. In addition to code explicitly declared to be strict, any code in a `class` body (Chapter 9) or in an ES6 module (§10.3) is automatically strict code. This means that if all of your JavaScript code is written as modules, then it is all automatically strict, and you will never need to use an explicit `"use strict"` directive.

Strict code is executed in *strict mode*. Strict mode is a restricted subset of the language that fixes important language deficiencies and provides stronger error checking and increased security. Because strict mode is not the default, old JavaScript code that still uses the deficient legacy features of the language will continue to run correctly. 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.) Also, in strict mode, when a function is invoked with `call()` or `apply()` (§8.7.4), the `this` value is exactly the value passed as the first argument to `call()` or `apply()`. (In non-strict mode, `null` and `undefined` values are replaced with the global object and nonobject values are converted to objects.)
- In strict mode, assignments to nonwritable properties and attempts to create new properties on non-extensible 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 (§8.3.3) 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 nonstrict 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.)

5.7 Declarations

The keywords `const`, `let`, `var`, `function`, `class`, `import`, and `export` are not technically statements, but they look a lot like statements, and this book refers informally to them as statements, so they deserve a mention in this chapter.

These keywords are more accurately described as *declarations* rather than statements. We said at the start of this chapter that statements "make something happen." Declarations serve to define new values and give them names that we can use to refer to those values. They don't make much happen themselves, but by providing names for values they, in an important sense, define the meaning of the other statements in your program.

When a program runs, it is the program's expressions that are being evaluated and the program's statements that are being executed. The declarations in a program don't "run" in the same way: instead, they define the structure of the program itself. Loosely, you can think of declarations as the parts of the program that are processed before the code starts running.

JavaScript declarations are used to define constants, variables, functions, and classes and for importing and exporting values between modules. The next subsections give examples of all of these declarations. They are all covered in much more detail elsewhere in this book.

5.7.1 const, let, and var

The `const`, `let`, and `var` declarations are covered in §3.10. In ES6 and later, `const` declares constants, and `let` declares variables. Prior to ES6, the `var` keyword was the only way to declare variables and there was no way to declare constants. Variables declared with `var` are scoped to the containing function rather than the containing block. This can be a source of bugs, and in modern JavaScript there is really no reason to use `var` instead of `let`.

```
const TAU = 2*Math.PI;
let radius = 3;
var circumference = TAU * radius;
```

5.7.2 function

The `function` declaration is used to define functions, which are covered in detail in Chapter 8. (We also saw `function` in §4.3, where it was used as part of a function expression rather than a function declaration.) A function declaration looks like this:

```
function area(radius) {
    return Math.PI * radius * radius;
}
```

A function declaration creates a function object and assigns it to the specified name—`area` in this example. Elsewhere in our program, we can refer to the function—and run the code inside it—by using this name. The function declarations in any block of JavaScript code are processed before that code runs, and the function names are bound to the function objects throughout the block. We say that function declarations are "hoisted" because it is as if they had all been moved up to the top of whatever scope they are defined within. The upshot is that code that invokes a function can exist in your program before the code that declares the function.

§12.3 describes a special kind of function known as a *generator*. Generator declarations use the `function` keyword but follow it with an asterisk. §13.3 describes asynchronous functions, which are also declared using the `function` keyword but are prefixed with the `async` keyword.

5.7.3 class

In ES6 and later, the `class` declaration creates a new class and gives it a name that we can use to refer to it. Classes are described in detail in Chapter 9. A simple class declaration might look like this:

```
class Circle {
    constructor(radius) { this.r = radius; }
    area() { return Math.PI * this.r * this.r; }
    circumference() { return 2 * Math.PI * this.r; }
}
```

Unlike functions, class declarations are not hoisted, and you cannot use a class declared this way in code that appears before the declaration.

5.7.4 import and export

The `import` and `export` declarations are used together to make values defined in one module of JavaScript code available in another module. A module is a file of JavaScript code with its own global namespace, completely independent of all other modules. The only way that a value (such as function or class) defined in one module can be used in another module is if the defining module exports it with `export` and the using module imports it with `import`. Modules are the subject of Chapter 10, and `import` and `export` are covered in detail in §10.3.

`import` directives are used to import one or more values from another file of JavaScript code and give them names within the current module. `import` directives come in a few different forms. Here are some examples:

```
import Circle from './geometry/circle.js';
import { PI, TAU } from './geometry/constants.js';
import { magnitude as hypotenuse } from './vectors/utils.js';
```

Values within a JavaScript module are private and cannot be imported into other modules unless they have been explicitly exported. The `export` directive does this: it declares that one or more values defined in the current module are exported and therefore available for import by other modules. The `export` directive has more variants than the `import` directive does. Here is one of them:

```
// geometry/constants.js
const PI = Math.PI;
const TAU = 2 * PI;
export { PI, TAU };
```

The `export` keyword is sometimes used as a modifier on other declarations, resulting in a kind of compound declaration that defines a constant, variable, function, or class and exports it at the same time. And when a module exports only a single value, this is typically done with the special form `export default`:

```
export const TAU = 2 * Math.PI;
export function magnitude(x,y) { return Math.sqrt(x*x + y*y); }
export default class Circle { /* class definition omitted here */ }
```

5.8 Summary of JavaScript Statements

This chapter introduced each of the JavaScript language's statements, which are summarized in Table 5-1.

Table 5-1. JavaScript statement syntax

Statement	Purpose
break	Exit from the innermost loop or `switch` or from named enclosing statement
case	Label a statement within a `switch`
class	Declare a class
const	Declare and initialize one or more constants
continue	Begin next iteration of the innermost loop or the named loop
debugger	Debugger breakpoint
default	Label the default statement within a `switch`
do/while	An alternative to the `while` loop
export	Declare values that can be imported into other modules
for	An easy-to-use loop
for/await	Asynchronously iterate the values of an async iterator
for/in	Enumerate the property names of an object
for/of	Enumerate the values of an iterable object such as an array
function	Declare a function
if/else	Execute one statement or another depending on a condition
import	Declare names for values defined in other modules
label	Give statement a name for use with `break` and `continue`
let	Declare and initialize one or more block-scoped variables (new syntax)
return	Return a value from a function
switch	Multiway branch to `case` or `default:` labels
throw	Throw an exception
try/catch/finally	Handle exceptions and code cleanup
"use strict"	Apply strict mode restrictions to script or function
var	Declare and initialize one or more variables (old syntax)
while	A basic loop construct
with	Extend the scope chain (deprecated and forbidden in strict mode)
yield	Provide a value to be iterated; only used in generator functions

CHAPTER 6

Objects

Objects are JavaScript's most fundamental datatype, and you have already seen them many times in the chapters that precede this one. Because objects are so important to the JavaScript language, it is important that you understand how they work in detail, and this chapter provides that detail. It begins with a formal overview of objects, then dives into practical sections about creating objects and querying, setting, deleting, testing, and enumerating the properties of objects. These property-focused sections are followed by sections that explain how to extend, serialize, and define important methods on objects. Finally, the chapter concludes with a long section about new object literal syntax in ES6 and more recent versions of the language.

6.1 Introduction to Objects

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 usually strings (although, as we'll see in §6.10.3, property names can also be Symbols), 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, a Symbol, or `true`, `false`, `null`, or `undefined` is an object. And even though strings, numbers, and booleans are not objects, they can behave like immutable objects.

Recall from §3.8 that objects are *mutable* and manipulated by reference rather than by value. If the variable `x` refers to an object and the code `let 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 to create them and set, query, delete, test, and enumerate their properties. These fundamental operations are described in the opening sections of this chapter. The sections after that cover more advanced topics.

A *property* has a name and a value. A property name may be any string, including the empty string (or any Symbol), but no object may have two properties with the same name. The value may be any JavaScript value, or it may be a getter or setter function (or both). We'll learn about getter and setter functions in §6.10.6.

It is sometimes important to be able to distinguish between properties defined directly on an object and those that are inherited from a prototype object. JavaScript uses the term *own property* to refer to non-inherited properties.

In addition to its name and value, each property has three *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.

Many of JavaScript's built-in objects have properties that are read-only, non-enumerable, or non-configurable. By default, however, all properties of the objects you create are writable, enumerable, and configurable. §14.1 explains techniques for specifying non-default property attribute values for your objects.

6.2 Creating Objects

Objects can be created with object literals, with the `new` keyword, and with the `Object.create()` function. The subsections below describe each technique.

6.2.1 Object Literals

The easiest way to create an object is to include an object literal in your JavaScript code. In its simplest form, 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:

```
let empty = {};                           // An object with no properties
let point = { x: 0, y: 0 };               // Two numeric properties
let p2 = { x: point.x, y: point.y+1 };    // More complex values
let book = {
    "main title": "JavaScript",           // These property names include spaces,
    "sub-title": "The Definitive Guide", // and hyphens, so use string literals.
    for: "all audiences",                 // for is reserved, but no quotes.
    author: {                             // The value of this property is
        firstname: "David",               // itself an object.
        surname: "Flanagan"
    }
};
```

A trailing comma following the last property in an object literal is legal, and some programming styles encourage the use of these trailing commas so you're less likely to cause a syntax error if you add a new property at the end of the object literal at some later time.

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 or in a function that is called repeatedly, and that the property values of these objects may differ from each other.

The object literals shown here use simple syntax that has been legal since the earliest versions of JavaScript. Recent versions of the language have introduced a number of new object literal features, which are covered in §6.10.

6.2.2 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. JavaScript includes constructors for its built-in types. For example:

```
let o = new Object();  // Create an empty object: same as {}.
let a = new Array();   // Create an empty array: same as [].
let d = new Date();    // Create a Date object representing the current time
let r = new Map();     // Create a Map object for key/value mapping
```

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.

6.2.3 Prototypes

Before we can cover the third object creation technique, we must pause for a moment to explain prototypes. Almost every JavaScript object has a second JavaScript object 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. This can be confusing when first learning JavaScript. Remember: almost all objects have a *prototype*, but only a relatively small number of objects have a `prototype` property. It is these objects with `prototype` properties that define the *prototypes* for all the other objects.

`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. Most 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 §6.3.2. 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. And we'll learn how to query (and even change) the prototype of an object in §14.3.

6.2.4 Object.create()

`Object.create()` creates a new object, using its first argument as the prototype of that object:

```
let o1 = Object.create({x: 1, y: 2});     // o1 inherits properties x and y.
o1.x + o1.y                               // => 3
```

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):

```
let 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`:

```
let o3 = Object.create(Object.prototype); // o3 is like {} or new Object().
```

The ability to create a new object with an arbitrary prototype is a powerful one, and we'll use `Object.create()` in a number of places throughout this chapter. (`Object.create()` also takes an optional second argument that describes the properties of the new object. This second argument is an advanced feature covered in §14.1.)

One use for `Object.create()` 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 object that inherits from it. If the function reads properties of that object, it will see the inherited values. If it sets properties, however, those writes will not affect the original object.

```
let o = { x: "don't change this value" };
library.function(Object.create(o));  // Guard against accidental modifications
```

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.

6.3 Querying and Setting Properties

To obtain the value of a property, use the dot (`.`) or square bracket (`[]`) operators described in §4.4. The lefthand side should be an expression whose value is an object. If using the dot operator, the righthand side 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:

```
let author = book.author;       // Get the "author" property of the book.
let name = author.surname;      // Get the "surname" property of the author.
let 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 lefthand side of an assignment expression:

```
book.edition = 7;                  // Create an "edition" property of book.
book["main title"] = "ECMAScript";  // Change the "main title" property.
```

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 or to a Symbol (§6.10.3). In Chapter 7, for example, we'll see that it is common to use numbers inside the square brackets.

6.3.1 Objects As Associative Arrays

As explained in the preceding section, 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:

```
let addr = "";
for(let 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. This code 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 they own 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.

In Chapter 5, we introduced the `for/in` loop (and we'll see it again shortly, in §6.6). The power of this JavaScript statement becomes clear when you consider its use with associative arrays. Here is how you would use it when computing the total value of a portfolio:

```
function computeValue(portfolio) {
    let total = 0.0;
    for(let stock in portfolio) {       // For each stock in the portfolio:
        let shares = portfolio[stock];  // get the number of shares
        let price = getQuote(stock);    // look up share price
        total += shares * price;        // add stock value to total value
    }
    return total;                       // Return total value.
}
```

JavaScript objects are commonly used as associative arrays as shown here, and it is important to understand how this works. In ES6 and later, however, the Map class described in §11.1.2 is often a better choice than using a plain object.

6.3.2 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 `Object.create()` function to create objects with specified prototypes. We'll see in Chapter 9, however, that every time you create an instance of a class with `new`, you are creating an object that inherits properties from a prototype object.

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[1] 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:

```
let o = {};              // o inherits object methods from Object.prototype
o.x = 1;                 // and it now has an own property x.
let p = Object.create(o); // p inherits properties from o and Object.prototype
p.y = 2;                 // and has an own property y.
let q = Object.create(p); // q inherits properties from p, o, and...
q.z = 3;                 // ...Object.prototype and has an own property z.
let f = 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 (non-inherited) 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 only 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 §6.3.3.) If the assignment is allowed, however, it always creates or sets a property in the original object and never modifies objects in 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:

```
let unitcircle = { r: 1 };         // An object to inherit from
let c = Object.create(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 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 §6.10.6), 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

1 Remember; almost all objects have a prototype but most do not have a property named `prototype`. JavaScript inheritance works even if you can't access the prototype object directly. But see §14.3 if you want to learn how to do that.

property, so if the setter method defines any properties, it will do so on `o`, and it will again leave the prototype chain unmodified.

6.3.3 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 preceding example:

```
let len = book.subtitle.length; // !TypeError: undefined doesn't have length
```

Property access expressions will fail if the lefthand side of the `.` is `null` or `undefined`. So when writing an expression like `book.author.surname`, you should be careful if you are not certain that `book` and `book.author` are actually defined. Here are two ways to guard against this kind of problem:

```
// A verbose and explicit technique
let surname = undefined;
if (book) {
    if (book.author) {
        surname = book.author.surname;
    }
}

// A concise and idiomatic alternative to get surname or null or undefined
surname = book && book.author && book.author.surname;
```

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.

As described in §4.4.1, ES2020 supports conditional property access with `?.`, which allows us to rewrite the previous assignment expression as:

```
let surname = book?.author?.surname;
```

Attempting to set a property on `null` or `undefined` also causes a TypeError. 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. In strict mode (§5.6.3), a TypeError is thrown whenever an attempt to set a property fails. Outside of strict mode, these failures are usually silent.

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.
- `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 §14.2) is `false`. Since `p` does not already exist in `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.

6.4 Deleting Properties

The `delete` operator (§4.13.4) 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:

```
let o = {x: 1};    // o has own property x and inherits property toString
delete o.x         // => true: deletes property x
delete o.x         // => true: does nothing (x doesn't exist) but true anyway
delete o.toString  // => true: does nothing (toString isn't an own property)
delete 1           // => true: nonsense, but true anyway
```

`delete` does not remove properties that have a *configurable* attribute of `false`. Certain properties of built-in objects are non-configurable, as are properties of the global object created by variable declaration and function declaration. In strict mode, attempting to delete a non-configurable property causes a TypeError. In non-strict mode, `delete` simply evaluates to `false` in this case:

```
// In strict mode, all these deletions throw TypeError instead of returning false
delete Object.prototype // => false: property is non-configurable
var x = 1;              // Declare a global variable
delete globalThis.x     // => false: can't delete this property
```

```
function f() {}         // Declare a global function
delete globalThis.f     // => false: 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:

```
globalThis.x = 1;       // Create a configurable global property (no let or var)
delete x                // => true: this property can be deleted
```

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 globalThis.x;    // This works
```

6.5 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 examples shown here all use strings as property names, but they also work with Symbols (§6.10.3).

The `in` operator expects a property name 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:

```
let 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:

```
let 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 techniques shown in §14.1 to make them non-enumerable.

```
let o = { x: 1 };
o.propertyIsEnumerable("x")  // => true: o has an own enumerable property x
```

```
o.propertyIsEnumerable("toString")  // => false: not an own property
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:

```
let 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 here cannot do. `in` can distinguish between properties that do not exist and properties that exist but have been set to `undefined`. Consider this code:

```
let 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 exist
delete o.x;                // Delete the property x
"x" in o                   // => false: it doesn't exist anymore
```

6.6 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. There are a few different ways to do this.

The `for/in` loop was covered in §5.4.5. 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 by default. For example:

```
let o = {x: 1, y: 2, z: 3};          // Three enumerable own properties
o.propertyIsEnumerable("toString")   // => false: not enumerable
for(let p in o) {                    // Loop through the properties
    console.log(p);                  // Prints x, y, and z, but not toString
}
```

To guard against enumerating inherited properties with `for/in`, you can add an explicit check inside the loop body:

```
for(let p in o) {
    if (!o.hasOwnProperty(p)) continue;       // Skip inherited properties
}

for(let p in o) {
    if (typeof o[p] === "function") continue; // Skip all methods
}
```

As an alternative to using a `for/in` loop, it is often easier to get an array of property names for an object and then loop through that array with a `for/of` loop. There are four functions you can use to get an array of property names:

- `Object.keys()` returns an array of the names of the enumerable own properties of an object. It does not include non-enumerable properties, inherited properties, or properties whose name is a Symbol (see §6.10.3).
- `Object.getOwnPropertyNames()` works like `Object.keys()` but returns an array of the names of non-enumerable own properties as well, as long as their names are strings.
- `Object.getOwnPropertySymbols()` returns own properties whose names are Symbols, whether or not they are enumerable.
- `Reflect.ownKeys()` returns all own property names, both enumerable and non-enumerable, and both string and Symbol. (See §14.6.)

There are examples of the use of `Object.keys()` with a `for/of` loop in §6.7.

6.6.1 Property Enumeration Order

ES6 formally defines the order in which the own properties of an object are enumerated. `Object.keys()`, `Object.getOwnPropertyNames()`, `Object.getOwnPropertySymbols()`, `Reflect.ownKeys()`, and related methods such as `JSON.stringify()` all list properties in the following order, subject to their own additional constraints about whether they list non-enumerable properties or properties whose names are strings or Symbols:

- String properties whose names are non-negative integers are listed first, in numeric order from smallest to largest. This rule means that arrays and array-like objects will have their properties enumerated in order.
- After all properties that look like array indexes are listed, all remaining properties with string names are listed (including properties that look like negative numbers or floating-point numbers). These properties are listed in the order in which they were added to the object. For properties defined in an object literal, this order is the same order they appear in the literal.
- Finally, the properties whose names are Symbol objects are listed in the order in which they were added to the object.

The enumeration order for the `for/in` loop is not as tightly specified as it is for these enumeration functions, but implementations typically enumerate own properties in the order just described, then travel up the prototype chain enumerating properties in the same order for each prototype object. Note, however, that a property will not be

enumerated if a property by that same name has already been enumerated, or even if a non-enumerable property by the same name has already been considered.

6.7 Extending Objects

A common operation in JavaScript programs is needing to copy the properties of one object to another object. It is easy to do that with code like this:

```
let target = {x: 1}, source = {y: 2, z: 3};
for(let key of Object.keys(source)) {
    target[key] = source[key];
}
target  // => {x: 1, y: 2, z: 3}
```

But because this is a common operation, various JavaScript frameworks have defined utility functions, often named `extend()`, to perform this copying operation. Finally, in ES6, this ability comes to the core JavaScript language in the form of `Object.assign()`.

`Object.assign()` expects two or more objects as its arguments. It modifies and returns the first argument, which is the target object, but does not alter the second or any subsequent arguments, which are the source objects. For each source object, it copies the enumerable own properties of that object (including those whose names are Symbols) into the target object. It processes the source objects in argument list order so that properties in the first source object override properties by the same name in the target object and properties in the second source object (if there is one) override properties with the same name in the first source object.

`Object.assign()` copies properties with ordinary property get and set operations, so if a source object has a getter method or the target object has a setter method, they will be invoked during the copy, but they will not themselves be copied.

One reason to assign properties from one object into another is when you have an object that defines default values for many properties and you want to copy those default properties into another object if a property by that name does not already exist in that object. Using `Object.assign()` naively will not do what you want:

```
Object.assign(o, defaults);  // overwrites everything in o with defaults
```

Instead, what you can do is to create a new object, copy the defaults into it, and then override those defaults with the properties in `o`:

```
o = Object.assign({}, defaults, o);
```

We'll see in §6.10.4 that you can also express this object copy-and-override operation using the `...` spread operator like this:

```
o = {...defaults, ...o};
```

We could also avoid the overhead of the extra object creation and copying by writing a version of `Object.assign()` that copies properties only if they are missing:

```
// Like Object.assign() but doesn't override existing properties
// (and also doesn't handle Symbol properties)
function merge(target, ...sources) {
    for(let source of sources) {
        for(let key of Object.keys(source)) {
            if (!(key in target)) { // This is different than Object.assign()
                target[key] = source[key];
            }
        }
    }
    return target;
}
Object.assign({x: 1}, {x: 2, y: 2}, {y: 3, z: 4})  // => {x: 2, y: 3, z: 4}
merge({x: 1}, {x: 2, y: 2}, {y: 3, z: 4})          // => {x: 1, y: 2, z: 4}
```

It is straightforward to write other property manipulation utilities like this `merge()` function. A `restrict()` function could delete properties of an object if they do not appear in another template object, for example. Or a `subtract()` function could remove all of the properties of one object from another object.

6.8 Serializing Objects

Object *serialization* is the process of converting an object's state to a string from which it can later be restored. The functions `JSON.stringify()` and `JSON.parse()` 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:

```
let o = {x: 1, y: {z: [false, null, ""]}}; // Define a test object
let s = JSON.stringify(o);   // s == '{"x":1,"y":{"z":[false,null,""]}}'
let p = JSON.parse(s);       // p == {x: 1, y: {z: [false, null, ""]}}
```

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 §11.6.

6.9 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()` and `propertyIsEnumerable()` methods, for example. (And we've also already covered quite a few static functions defined on the `Object` constructor, such as `Object.create()` and `Object.keys()`.) This section explains a handful of universal object methods that are defined on `Object.prototype`, but which are intended to be replaced by other, more specialized implementations. In the sections that follow, we show examples of defining these methods on a single object. In Chapter 9, you'll learn how to define these methods more generally for an entire class of objects.

6.9.1 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 will see in §14.4.3). For example, the following line of code simply evaluates to the string "[object Object]":

```
let s = { x: 1, y: 1 }.toString();  // s == "[object Object]"
```

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. You might define your own `toString()` method like this:

```
let point = {
    x: 1,
    y: 2,
    toString: function() { return `(${this.x}, ${this.y})`; }
};
String(point)    // => "(1, 2)": toString() is used for string conversions
```

6.9.2 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. You might do the same thing with a `point` object like this:

```
let point = {
    x: 1000,
    y: 2000,
    toString: function() { return `(${this.x}, ${this.y})`; },
    toLocaleString: function() {
        return `(${this.x.toLocaleString()}, ${this.y.toLocaleString()})`;
    }
};
point.toString()        // => "(1000, 2000)"
point.toLocaleString()  // => "(1,000, 2,000)": note thousands separators
```

The internationalization classes documented in §11.7 can be useful when implementing a `toLocaleString()` method.

6.9.3 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. The Date class defines `valueOf()` to convert dates to numbers, and this allows Date objects to be chronologically compared with < and >. You could do something similar with a point object, defining a `valueOf()` method that returns the distance from the origin to the point:

```
let point = {
    x: 3,
    y: 4,
    valueOf: function() { return Math.hypot(this.x, this.y); }
};
Number(point)  // => 5: valueOf() is used for conversions to numbers
point > 4      // => true
point > 5      // => false
point < 6      // => true
```

6.9.4 The toJSON() Method

`Object.prototype` does not actually define a `toJSON()` method, but the `JSON.stringify()` method (see §6.8) 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. The Date class (§11.4) defines a `toJSON()` method that returns a serializable string representation of the date. We could do the same for our Point object like this:

```
let point = {
    x: 1,
    y: 2,
    toString: function() { return `(${this.x}, ${this.y})`; },
    toJSON: function() { return this.toString(); }
};
JSON.stringify([point])   // => '["(1, 2)"]'
```

6.10 Extended Object Literal Syntax

Recent versions of JavaScript have extended the syntax for object literals in a number of useful ways. The following subsections explain these extensions.

6.10.1 Shorthand Properties

Suppose you have values stored in variables `x` and `y` and want to create an object with properties named `x` and `y` that hold those values. With basic object literal syntax, you'd end up repeating each identifier twice:

```
let x = 1, y = 2;
let o = {
    x: x,
    y: y
};
```

In ES6 and later, you can drop the colon and one copy of the identifier and end up with much simpler code:

```
let x = 1, y = 2;
let o = { x, y };
o.x + o.y  // => 3
```

6.10.2 Computed Property Names

Sometimes you need to create an object with a specific property, but the name of that property is not a compile-time constant that you can type literally in your source code. Instead, the property name you need is stored in a variable or is the return value of a function that you invoke. You can't use a basic object literal for this kind of property. Instead, you have to create an object and then add the desired properties as an extra step:

```
const PROPERTY_NAME = "p1";
function computePropertyName() { return "p" + 2; }

let o = {};
o[PROPERTY_NAME] = 1;
o[computePropertyName()] = 2;
```

It is much simpler to set up an object like this with an ES6 feature known as *computed properties* that lets you take the square brackets from the preceding code and move them directly into the object literal:

```
const PROPERTY_NAME = "p1";
function computePropertyName() { return "p" + 2; }

let p = {
    [PROPERTY_NAME]: 1,
    [computePropertyName()]: 2
};

p.p1 + p.p2 // => 3
```

With this new syntax, the square brackets delimit an arbitrary JavaScript expression. That expression is evaluated, and the resulting value (converted to a string, if necessary) is used as the property name.

One situation where you might want to use computed properties is when you have a library of JavaScript code that expects to be passed objects with a particular set of properties, and the names of those properties are defined as constants in that library. If you are writing code to create the objects that will be passed to that library, you could hardcode the property names, but you'd risk bugs if you type the property name wrong anywhere, and you'd risk version mismatch issues if a new version of the library changes the required property names. Instead, you might find that it makes your code more robust to use computed property syntax with the property name constants defined by the library.

6.10.3 Symbols as Property Names

The computed property syntax enables one other very important object literal feature. In ES6 and later, property names can be strings or symbols. If you assign a symbol to a variable or constant, then you can use that symbol as a property name using the computed property syntax:

```
const extension = Symbol("my extension symbol");
let o = {
    [extension]: { /* extension data stored in this object */ }
};
o[extension].x = 0; // This won't conflict with other properties of o
```

As explained in §3.6, Symbols are opaque values. You can't do anything with them other than use them as property names. Every Symbol is different from every other Symbol, however, which means that Symbols are good for creating unique property names. Create a new Symbol by calling the `Symbol()` factory function. (Symbols are primitive values, not objects, so `Symbol()` is not a constructor function that you invoke with `new`.) The value returned by `Symbol()` is not equal to any other Symbol or other value. You can pass a string to `Symbol()`, and this string is used when your Symbol is converted to a string. But this is a debugging aid only: two Symbols created with the same string argument are still different from one another.

The point of Symbols is not security, but to define a safe extension mechanism for JavaScript objects. If you get an object from third-party code that you do not control and need to add some of your own properties to that object but want to be sure that your properties will not conflict with any properties that may already exist on the object, you can safely use Symbols as your property names. If you do this, you can also be confident that the third-party code will not accidentally alter your symbolically named properties. (That third-party code could, of course, use `Object.getOwnPropertySymbols()` to discover the Symbols you're using and could then alter or delete your properties. This is why Symbols are not a security mechanism.)

6.10.4 Spread Operator

In ES2018 and later, you can copy the properties of an existing object into a new object using the "spread operator" `...` inside an object literal:

```
let position = { x: 0, y: 0 };
let dimensions = { width: 100, height: 75 };
let rect = { ...position, ...dimensions };
rect.x + rect.y + rect.width + rect.height // => 175
```

In this code, the properties of the `position` and `dimensions` objects are "spread out" into the `rect` object literal as if they had been written literally inside those curly braces. Note that this `...` syntax is often called a spread operator but is not a true JavaScript operator in any sense. Instead, it is a special-case syntax available only

within object literals. (Three dots are used for other purposes in other JavaScript contexts, but object literals are the only context where the three dots cause this kind of interpolation of one object into another one.)

If the object that is spread and the object it is being spread into both have a property with the same name, then the value of that property will be the one that comes last:

```
let o = { x: 1 };
let p = { x: 0, ...o };
p.x   // => 1: the value from object o overrides the initial value
let q = { ...o, x: 2 };
q.x   // => 2: the value 2 overrides the previous value from o.
```

Also note that the spread operator only spreads the own properties of an object, not any inherited ones:

```
let o = Object.create({x: 1}); // o inherits the property x
let p = { ...o };
p.x                            // => undefined
```

Finally, it is worth noting that, although the spread operator is just three little dots in your code, it can represent a substantial amount of work to the JavaScript interpreter. If an object has n properties, the process of spreading those properties into another object is likely to be an $O(n)$ operation. This means that if you find yourself using `...` within a loop or recursive function as a way to accumulate data into one large object, you may be writing an inefficient $O(n^2)$ algorithm that will not scale well as n gets larger.

6.10.5 Shorthand Methods

When a function is defined as a property of an object, we call that function a *method* (we'll have a lot more to say about methods in Chapters 8 and 9). Prior to ES6, you would define a method in an object literal using a function definition expression just as you would define any other property of an object:

```
let square = {
    area: function() { return this.side * this.side; },
    side: 10
};
square.area() // => 100
```

In ES6, however, the object literal syntax (and also the class definition syntax we'll see in Chapter 9) has been extended to allow a shortcut where the `function` keyword and the colon are omitted, resulting in code like this:

```
let square = {
    area() { return this.side * this.side; },
    side: 10
};
square.area() // => 100
```

Both forms of the code are equivalent: both add a property named `area` to the object literal, and both set the value of that property to the specified function. The shorthand syntax makes it clearer that `area()` is a method and not a data property like `side`.

When you write a method using this shorthand syntax, the property name can take any of the forms that are legal in an object literal: in addition to a regular JavaScript identifier like the name `area` above, you can also use string literals and computed property names, which can include Symbol property names:

```
const METHOD_NAME = "m";
const symbol = Symbol();
let weirdMethods = {
    "method With Spaces"(x) { return x + 1; },
    [METHOD_NAME](x) { return x + 2; },
    [symbol](x) { return x + 3; }
};
weirdMethods["method With Spaces"](1)  // => 2
weirdMethods[METHOD_NAME](1)           // => 3
weirdMethods[symbol](1)                // => 4
```

Using a Symbol as a method name is not as strange as it seems. In order to make an object iterable (so it can be used with a `for/of` loop), you must define a method with the symbolic name `Symbol.iterator`, and there are examples of doing exactly that in Chapter 12.

6.10.6 Property Getters and Setters

All of the object properties we've discussed so far in this chapter have been *data properties* with a name and an ordinary value. JavaScript also supports *accessor properties*, which do not have a single value but instead have one or two accessor methods: a *getter* and/or a *setter*.

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 righthand 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.

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`.

Accessor properties can be defined with an extension to the object literal syntax (unlike the other ES6 extensions we've seen here, getters and setters were introduced in ES5):

```
let o = {
    // An ordinary data property
    dataProp: value,

    // An accessor property defined as a pair of functions.
    get accessorProp() { return this.dataProp; },
    set accessorProp(value) { this.dataProp = value; }
};
```

Accessor properties are defined as one or two methods whose name is the same as the property name. These look like ordinary methods defined using the ES6 shorthand except that getter and setter definitions are prefixed with `get` or `set`. (In ES6, you can also use computed property names when defining getters and setters. Simply replace the property name after `get` or `set` with an expression in square brackets.)

The accessor methods defined above simply get and set the value of a data property, and there is no reason to prefer the accessor property over the data property. But as a more interesting 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 that give the equivalent polar coordinates of the point:

```
let 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.hypot(this.x, this.y); },
    set r(newvalue) {
        let oldvalue = Math.hypot(this.x, this.y);
        let 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); }
};
p.r     // => Math.SQRT2
p.theta // => Math.PI / 4
```

Note the use of the keyword `this` in the getters and setter in this example. 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 `p`. 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:

```
let q = Object.create(p); // A new object that inherits getters and setters
q.x = 3; q.y = 4;         // Create q's own data properties
q.r                       // => 5: the inherited accessor properties work
q.theta                   // => Math.atan2(4, 3)
```

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
const serialnum = {
    // This data property holds the next serial number.
    // The _ in the property name hints that it is for internal use only.
    _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 new Error("serial number can only be set to a larger value");
    }
};
serialnum.next = 10;    // Set the starting serial number
serialnum.next          // => 10
serialnum.next          // => 11: different value each time we get next
```

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.
const 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; }
};
```

6.11 Summary

This chapter has documented JavaScript objects in great detail, covering topics that include:

- Basic object terminology, including the meaning of terms like *enumerable* and *own property*.
- Object literal syntax, including the many new features in ES6 and later.
- How to read, write, delete, enumerate, and check for the presence of the properties of an object.
- How prototype-based inheritance works in JavaScript and how to create an object that inherits from another object with `Object.create()`.
- How to copy properties from one object into another with `Object.assign()`.

All JavaScript values that are not primitive values are objects. This includes both arrays and functions, which are the topics of the next two chapters.

CHAPTER 7
Arrays

This chapter documents arrays, a fundamental datatype in JavaScript and in most other programming languages. 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 highest index of any element.

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. 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 §7.9. Finally, JavaScript strings behave like arrays of characters, and we'll discuss this in §7.10.

ES6 introduces a set of new array classes known collectively as "typed arrays." Unlike regular JavaScript arrays, typed arrays have a fixed length and a fixed numeric element type. They offer high performance and byte-level access to binary data and are covered in §11.2.

7.1 Creating Arrays

There are several ways to create arrays. The subsections that follow explain how to create arrays with:

- Array literals
- The `...` spread operator on an iterable object
- The `Array()` constructor
- The `Array.of()` and `Array.from()` factory methods

7.1.1 Array Literals

By far the simplest 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:

```
let empty = [];                 // An array with no elements
let primes = [2, 3, 5, 7, 11];  // An array with 5 numeric elements
let 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:

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

Array literals can contain object literals or other array literals:

```
let 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:

```
let count = [1,,3]; // Elements at indexes 0 and 2. No element at index 1
let undefs = [,,];  // An array with no elements but a length of 2
```

Array literal syntax allows an optional trailing comma, so `[,,]` has a length of 2, not 3.

7.1.2 The Spread Operator

In ES6 and later, you can use the "spread operator," `...`, to include the elements of one array within an array literal:

```
let a = [1, 2, 3];
let b = [0, ...a, 4];  // b == [0, 1, 2, 3, 4]
```

The three dots "spread" the array `a` so that its elements become elements within the array literal that is being created. It is as if the `...a` was replaced by the elements of the array `a`, listed literally as part of the enclosing array literal. (Note that, although we call these three dots a spread operator, this is not a true operator because it can only be used in array literals and, as we'll see later in the book, function invocations.)

The spread operator is a convenient way to create a (shallow) copy of an array:

```
let original = [1,2,3];
let copy = [...original];
copy[0] = 0;  // Modifying the copy does not change the original
original[0]   // => 1
```

The spread operator works on any iterable object. (*Iterable* objects are what the `for/of` loop iterates over; we first saw them in §5.4.4, and we'll see much more about them in Chapter 12.) Strings are iterable, so you can use a spread operator to turn any string into an array of single-character strings:

```
let digits = [..."0123456789ABCDEF"];
digits // => ["0","1","2","3","4","5","6","7","8","9","A","B","C","D","E","F"]
```

Set objects (§11.1.1) are iterable, so an easy way to remove duplicate elements from an array is to convert the array to a set and then immediately convert the set back to an array using the spread operator:

```
let letters = [..."hello world"];
[...new Set(letters)]  // => ["h","e","l","o"," ","w","r","d"]
```

7.1.3 The Array() Constructor

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:

  ```
  let 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:

  ```
  let 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:

```
let 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.

7.1.4 Array.of()

When the `Array()` constructor function is invoked with one numeric argument, it uses that argument as an array length. But when invoked with more than one numeric argument, it treats those arguments as elements for the array to be created. This means that the `Array()` constructor cannot be used to create an array with a single numeric element.

In ES6, the `Array.of()` function addresses this problem: it is a factory method that creates and returns a new array, using its argument values (regardless of how many of them there are) as the array elements:

```
Array.of()        // => []; returns empty array with no arguments
Array.of(10)      // => [10]; can create arrays with a single numeric argument
Array.of(1,2,3)   // => [1, 2, 3]
```

7.1.5 Array.from()

`Array.from` is another array factory method introduced in ES6. It expects an iterable or array-like object as its first argument and returns a new array that contains the elements of that object. With an iterable argument, `Array.from(iterable)` works like the spread operator `[...iterable]` does. It is also a simple way to make a copy of an array:

```
let copy = Array.from(original);
```

`Array.from()` is also important because it defines a way to make a true-array copy of an array-like object. Array-like objects are non-array objects that have a numeric length property and have values stored with properties whose names happen to be integers. When working with client-side JavaScript, the return values of some web browser methods are array-like, and it can be easier to work with them if you first convert them to true arrays:

```
let truearray = Array.from(arraylike);
```

`Array.from()` also accepts an optional second argument. If you pass a function as the second argument, then as the new array is being built, each element from the source object will be passed to the function you specify, and the return value of the function will be stored in the array instead of the original value. (This is very much like the array `map()` method that will be introduced later in the chapter, but it is more efficient to perform the mapping while the array is being built than it is to build the array and then map it to another new array.)

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:

```
let a = ["world"];     // Start with a one-element array
let value = a[0];      // Read element 0
a[1] = 3.14;           // Write element 1
let 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
```

What is special about arrays is that when you use property names that are non-negative integers less than $2^{32}-1$, the array automatically maintains the value of the `length` property for you. In the preceding, 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
```

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:

```
let o = {};    // Create a plain object
o[1] = "one";  // Index it with an integer
o["1"]         // => "one"; numeric and string property names are the same
```

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] = 1;     // Array index 1. Same as a[1] = 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:

```
let 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.
```

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`.

```
let 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:

```
let a1 = [,];           // This array has no elements and length 1
let 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
```

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 we 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.

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:

```
let 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:

```
let 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. The `pop()` method is the opposite of `push()`: it removes the last element of the array and returns it, reducing the length of an array by 1. Similarly, the `shift()` method removes and returns the first element of the array, reducing the length by 1 and shifting all elements down to an index one lower than their current index. See §7.8 for more on these methods.

You can delete array elements with the `delete` operator, just as you can delete object properties:

```
let a = [1,2,3];
delete a[2];   // a now has no element at index 2
2 in a         // => false: no array index 2 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 remove elements from the end of an array simply by setting the `length` property to the new desired length.

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

As of ES6, the easiest way to loop through each of the elements of an array (or any iterable object) is with the `for/of` loop, which was covered in detail in §5.4.4:

```
let letters = [..."Hello world"];  // An array of letters
let string = "";
for(let letter of letters) {
    string += letter;
}
string  // => "Hello world"; we reassembled the original text
```

The built-in array iterator that the `for/of` loop uses returns the elements of an array in ascending order. It has no special behavior for sparse arrays and simply returns `undefined` for any array elements that do not exist.

If you want to use a `for/of` loop for an array and need to know the index of each array element, use the `entries()` method of the array, along with destructuring assignment, like this:

```
let everyother = "";
for(let [index, letter] of letters.entries()) {
    if (index % 2 === 0) everyother += letter;  // letters at even indexes
}
everyother  // => "Hlowrd"
```

Another good way to iterate arrays is with `forEach()`. This is not a new form of the `for` loop, but an array method that offers a functional approach to array iteration. You pass a function to the `forEach()` method of an array, and `forEach()` invokes your function once on each element of the array:

```
let uppercase = "";
letters.forEach(letter => {  // Note arrow function syntax here
    uppercase += letter.toUpperCase();
});
uppercase  // => "HELLO WORLD"
```

As you would expect, `forEach()` iterates the array in order, and it actually passes the array index to your function as a second argument, which is occasionally useful. Unlike the `for/of` loop, the `forEach()` is aware of sparse arrays and does not invoke your function for elements that are not there.

§7.8.1 documents the `forEach()` method in more detail. That section also covers related methods such as `map()` and `filter()` that perform specialized kinds of array iteration.

You can also loop through the elements of an array with a good old-fashioned `for` loop (§5.4.3):

```
let vowels = "";
for(let i = 0; i < letters.length; i++) { // For each index in the array
    let letter = letters[i];              // Get the element at that index
    if (/[aeiou]/.test(letter)) {         // Use a regular expression test
        vowels += letter;                 // If it is a vowel, remember it
    }
}
vowels  // => "eoo"
```

In nested loops, or other contexts where performance is critical, you may sometimes see this basic array iteration loop written so that the array length is only looked up once rather than on each iteration. Both of the following `for` loop forms are

idiomatic, though not particularly common, and with modern JavaScript interpreters, it is not at all clear that they have any performance impact:

```
// Save the array length into a local variable
for(let i = 0, len = letters.length; i < len; i++) {
    // loop body remains the same
}

// Iterate backwards from the end of the array to the start
for(let i = letters.length-1; i >= 0; 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 skip undefined and nonexistent elements, you might write:

```
for(let i = 0; i < a.length; i++) {
    if (a[i] === undefined) continue; // Skip undefined + nonexistent elements
    // loop body here
}
```

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
let table = new Array(10);              // 10 rows of the table
for(let i = 0; i < table.length; i++) {
    table[i] = new Array(10);           // Each row has 10 columns
}

// Initialize the array
for(let row = 0; row < table.length; row++) {
    for(let col = 0; col < table[row].length; col++) {
        table[row][col] = row*col;
    }
}

// Use the multidimensional array to compute 5*7
table[5][7]  // => 35
```

7.8 Array Methods

The preceding sections have focused on basic JavaScript syntax for working with arrays. In general, though, it is the methods defined by the Array class that are the most powerful. The next sections document these methods. While reading about these methods, keep in mind that some of them modify the array they are called on and some of them leave the array unchanged. A number of the methods return an array: sometimes, this is a new array, and the original is unchanged. Other times, a method will modify the array in place and will also return a reference to the modified array.

Each of the subsections that follows covers a group of related array methods:

- Iterator methods loop over the elements of an array, typically invoking a function that you specify on each of those elements.
- Stack and queue methods add and remove array elements to and from the beginning and the end of an array.
- Subarray methods are for extracting, deleting, inserting, filling, and copying contiguous regions of a larger array.
- Searching and sorting methods are for locating elements within an array and for sorting the elements of an array.

The following subsections also cover the static methods of the Array class and a few miscellaneous methods for concatenating arrays and converting arrays to strings.

7.8.1 Array Iterator Methods

The methods described in this section iterate over arrays by passing array elements, in order, to a function you supply, and they provide convenient ways to iterate, map, filter, test, and reduce arrays.

Before we explain the methods in detail, however, it is worth making some generalizations about them. First, all of these 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 iterator methods described in the following subsections 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 as the first argument. The return value of the function you pass is usually important, but different methods handle the

return value in different ways. None of the methods described here modify the array on which they are invoked (though the function you pass can modify the array, of course).

Each of these functions is invoked with a function as its first argument, and it is very common to define that function inline as part of the method invocation expression instead of using an existing function that is defined elsewhere. Arrow function syntax (see §8.1.3) works particularly well with these methods, and we will use it in the examples that follow.

forEach()

The `forEach()` method iterates through an array, invoking a function you specify for each element. As we've described, 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:

```
let data = [1,2,3,4,5], sum = 0;
// Compute the sum of the elements of the array
data.forEach(value => { sum += value; });          // 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.

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 your function. For example:

```
let a = [1, 2, 3];
a.map(x => x*x)   // => [1, 4, 9]: the function takes input x and returns x*x
```

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, your function will not be called for the missing elements, but the returned array will be sparse in the same way as the original array: it will have the same length and the same missing elements.

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:

```
let a = [5, 4, 3, 2, 1];
a.filter(x => x < 3)         // => [2, 1]; values less than 3
a.filter((x,i) => i%2 === 0) // => [5, 3, 1]; every other value
```

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:

```
let dense = sparse.filter(() => true);
```

And to close gaps and remove undefined and null elements, you can use `filter`, like this:

```
a = a.filter(x => x !== undefined && x !== null);
```

find() and findIndex()

The `find()` and `findIndex()` methods are like `filter()` in that they iterate through your array looking for elements for which your predicate function returns a truthy value. Unlike `filter()`, however, these two methods stop iterating the first time the predicate finds an element. When that happens, `find()` returns the matching element, and `findIndex()` returns the index of the matching element. If no matching element is found, `find()` returns `undefined` and `findIndex()` returns `-1`:

```
let a = [1,2,3,4,5];
a.findIndex(x => x === 3)  // => 2; the value 3 appears at index 2
a.findIndex(x => x < 0)    // => -1; no negative numbers in the array
a.find(x => x % 5 === 0)   // => 5: this is a multiple of 5
a.find(x => x % 7 === 0)   // => undefined: no multiples of 7 in the array
```

every() and some()

The `every()` and `some()` methods are array predicates: they apply a predicate function you specify to the elements of the array, 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:

```
let a = [1,2,3,4,5];
a.every(x => x < 10)      // => true: all values are < 10.
a.every(x => x % 2 === 0) // => false: not all values are 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:

```
let a = [1,2,3,4,5];
a.some(x => 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 <code>true</code> 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.

reduce() and 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:

```
let a = [1,2,3,4,5];
a.reduce((x,y) => x+y, 0)          // => 15; the sum of the values
a.reduce((x,y) => x*y, 1)          // => 120; the product of the values
a.reduce((x,y) => (x > y) ? x : y) // => 5; the largest of the values
```

`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 we've shown here, 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, 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 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()` in this example 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, 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 associativity, for example:

```
// Compute 2^(3^4).  Exponentiation has right-to-left precedence
let a = [2, 3, 4];
a.reduceRight((acc,val) => Math.pow(val,acc)) // => 2.4178516392292583e+24
```

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 (§8.7.5) if you need your reduction function invoked as a method of a particular object.

The examples shown so far have been numeric for simplicity, but `reduce()` and `reduceRight()` are not intended solely for mathematical computations. Any function that can combine two values (such as two objects) into one value of the same type can be used as a reduction function. On the other hand, algorithms expressed using array reductions can quickly become complex and hard to understand, and you may find that it is easier to read, write, and reason about your code if you use regular looping constructs to process your arrays.

7.8.2 Flattening arrays with flat() and flatMap()

In ES2019, the `flat()` method creates and returns a new array that contains the same elements as the array it is called on, except that any elements that are themselves arrays are "flattened" into the returned array. For example:

```
[1, [2, 3]].flat()    // => [1, 2, 3]
[1, [2, [3]]].flat()  // => [1, 2, [3]]
```

When called with no arguments, `flat()` flattens one level of nesting. Elements of the original array that are themselves arrays are flattened, but array elements of *those* arrays are not flattened. If you want to flatten more levels, pass a number to `flat()`:

```
let a = [1, [2, [3, [4]]]];
a.flat(1)   // => [1, 2, [3, [4]]]
a.flat(2)   // => [1, 2, 3, [4]]
a.flat(3)   // => [1, 2, 3, 4]
a.flat(4)   // => [1, 2, 3, 4]
```

The `flatMap()` method works just like the `map()` method (see "map()" on page 166) except that the returned array is automatically flattened as if passed to `flat()`. That is, calling `a.flatMap(f)` is the same as (but more efficient than) `a.map(f).flat()`:

```
let phrases = ["hello world", "the definitive guide"];
let words = phrases.flatMap(phrase => phrase.split(" "));
words // => ["hello", "world", "the", "definitive", "guide"];
```

You can think of `flatMap()` as a generalization of `map()` that allows each element of the input array to map to any number of elements of the output array. In particular, `flatMap()` allows you to map input elements to an empty array, which flattens to nothing in the output array:

```
// Map non-negative numbers to their square roots
[-2, -1, 1, 2].flatMap(x => x < 0 ? [] : Math.sqrt(x)) // => [1, 2**0.5]
```

7.8.3 Adding arrays with concat()

The `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:

```
let a = [1,2,3];
a.concat(4, 5)          // => [1,2,3,4,5]
a.concat([4,5],[6,7])   // => [1,2,3,4,5,6,7]; arrays are flattened
a.concat(4, [5,[6,7]])  // => [1,2,3,4,5,[6,7]]; but not nested arrays
a                       // => [1,2,3]; the original array is unmodified
```

Note that `concat()` makes a new copy of the array it is called on. In many cases, this is the right thing to do, but it is an expensive operation. If you find yourself writing code like `a = a.concat(x)`, then you should think about modifying your array in place with `push()` or `splice()` instead of creating a new one.

7.8.4 Stacks and Queues with push(), pop(), shift(), and unshift()

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. Unlike `concat()`, `push()` does not flatten array arguments. 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. The combination of `push()` and `pop()` allows you to use a JavaScript array to implement a first-in, last-out stack. For example:

```
let stack = [];       // stack == []
stack.push(1,2);      // stack == [1,2];
stack.pop();          // stack == [1]; returns 2
stack.push(3);        // stack == [1,3]
stack.pop();          // stack == [1]; returns 3
stack.push([4,5]);    // stack == [1,[4,5]]
stack.pop()           // stack == [1]; returns [4,5]
stack.pop();          // stack == []; returns 1
```

The `push()` method does not flatten an array you pass to it, but if you want to push all of the elements from one array onto another array, you can use the spread operator (§8.3.4) to flatten it explicitly:

```
a.push(...values);
```

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. You could use `unshift()` and `shift()` to implement a stack, but it would be less efficient than using `push()` and `pop()` because the array elements need to be shifted up or down every time an element is added or removed at the start of the array. Instead, though, you can implement a queue data structure by using `push()` to add elements at the end of an array and `shift()` to remove them from the start of the array:

```
let q = [];           // q == []
q.push(1,2);          // q == [1,2]
q.shift();            // q == [2]; returns 1
q.push(3)             // q == [2, 3]
q.shift()             // q == [3]; returns 2
q.shift()             // q == []; returns 3
```

There is one feature of `unshift()` that is worth calling out because you may find it surprising. When passing multiple arguments to `unshift()`, they are inserted all at once, which means that they end up in the array in a different order than they would be if you inserted them one at a time:

```
let a = [];           // a == []
a.unshift(1)          // a == [1]
a.unshift(2)          // a == [2, 1]
a = [];               // a == []
a.unshift(1,2)        // a == [1, 2]
```

7.8.5 Subarrays with slice(), splice(), fill(), and copyWithin()

Arrays define a number of methods that work on contiguous regions, or subarrays or "slices" of an array. The following sections describe methods for extracting, replacing, filling, and copying slices.

slice()

The `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 length of the array. An argument of –1, for example, specifies the last element in the array, and an argument of –2 specifies the element before that one. Note that `slice()` does not modify the array on which it is invoked. Here are some examples:

```
let 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]
```

splice()

`splice()` 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. (Note that this is another difference between these two methods. The second argument to `slice()` is an end position. The second argument to `splice()` is a length.) 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:

```
let a = [1,2,3,4,5,6,7,8];
a.splice(4)    // => [5,6,7,8]; a is now [1,2,3,4]
```

```
a.splice(1,2)  // => [2,3]; a is now [1,4]
a.splice(1,1)  // => [4]; a is now [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:

```
let a = [1,2,3,4,5];
a.splice(2,0,"a","b")  // => []; a is now [1,2,"a","b",3,4,5]
a.splice(2,2,[1,2],3)  // => ["a","b"]; a is now [1,2,[1,2],3,3,4,5]
```

Note that, unlike `concat()`, `splice()` inserts arrays themselves, not the elements of those arrays.

fill()

The `fill()` method sets the elements of an array, or a slice of an array, to a specified value. It mutates the array it is called on, and also returns the modified array:

```
let a = new Array(5);   // Start with no elements and length 5
a.fill(0)               // => [0,0,0,0,0]; fill the array with zeros
a.fill(9, 1)            // => [0,9,9,9,9]; fill with 9 starting at index 1
a.fill(8, 2, -1)        // => [0,9,8,8,9]; fill with 8 at indexes 2, 3
```

The first argument to `fill()` is the value to set array elements to. The optional second argument specifies the starting index. If omitted, filling starts at index 0. The optional third argument specifies the ending index—array elements up to, but not including, this index will be filled. If this argument is omitted, then the array is filled from the start index to the end. You can specify indexes relative to the end of the array by passing negative numbers, just as you can for `slice()`.

copyWithin()

`copyWithin()` copies a slice of an array to a new position within the array. It modifies the array in place and returns the modified array, but it will not change the length of the array. The first argument specifies the destination index to which the first element will be copied. The second argument specifies the index of the first element to be copied. If this second argument is omitted, 0 is used. The third argument specifies the end of the slice of elements to be copied. If omitted, the length of the array is used. Elements from the start index up to, but not including, the end index will be copied. You can specify indexes relative to the end of the array by passing negative numbers, just as you can for `slice()`:

```
let a = [1,2,3,4,5];
a.copyWithin(1)       // => [1,1,2,3,4]: copy array elements up one
a.copyWithin(2, 3, 5) // => [1,1,3,4,4]: copy last 2 elements to index 2
a.copyWithin(0, -2)   // => [4,4,3,4,4]: negative offsets work, too
```

`copyWithin()` is intended as a high-performance method that is particularly useful with typed arrays (see §11.2). It is modeled after the `memmove()` function from the C standard library. Note that the copy will work correctly even if there is overlap between the source and destination regions.

7.8.6 Array Searching and Sorting Methods

Arrays implement `indexOf()`, `lastIndexOf()`, and `includes()` methods that are similar to the same-named methods of strings. There are also `sort()` and `reverse()` methods for reordering the elements of an array. These methods are described in the subsections that follow.

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:

```
let 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
```

`indexOf()` and `lastIndexOf()` compare their argument to the array elements using the equivalent of the `===` operator. If your array contains objects instead of primitive values, these methods check to see if two references both refer to exactly the same object. If you want to actually look at the content of an object, try using the `find()` method with your own custom predicate function instead.

`indexOf()` and `lastIndexOf()` take an optional second argument that 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 `slice()` 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) {
    let 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, except that a negative second argument is treated as zero.

includes()

The ES2016 `includes()` method takes a single argument and returns `true` if the array contains that value or `false` otherwise. It does not tell you the index of the value, only whether it exists. The `includes()` method is effectively a set membership test for arrays. Note, however, that arrays are not an efficient representation for sets, and if you are working with more than a few elements, you should use a real Set object (§11.1.1).

The `includes()` method is slightly different than the `indexOf()` method in one important way. `indexOf()` tests equality using the same algorithm that the `===` operator does, and that equality algorithm considers the not-a-number value to be different from every other value, including itself. `includes()` uses a slightly different version of equality that does consider `NaN` to be equal to itself. This means that `indexOf()` will not detect the `NaN` value in an array, but `includes()` will:

```
let a = [1,true,3,NaN];
a.includes(true)            // => true
a.includes(2)               // => false
a.includes(NaN)             // => true
a.indexOf(NaN)              // => -1; indexOf can't find NaN
```

sort()

`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):

```
let a = ["banana", "cherry", "apple"];
a.sort(); // a == ["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:

```
let a = [33, 4, 1111, 222];
a.sort();               // a == [1111, 222, 33, 4]; alphabetical order
a.sort(function(a,b) {  // Pass a comparator function
    return a-b;         // Returns < 0, 0, or > 0, depending on order
});                     // a == [4, 33, 222, 1111]; numerical order
a.sort((a,b) => b-a);   // a == [1111, 222, 33, 4]; reverse numerical order
```

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:

```
let a = ["ant", "Bug", "cat", "Dog"];
a.sort();    // a == ["Bug","Dog","ant","cat"]; case-sensitive sort
a.sort(function(s,t) {
    let a = s.toLowerCase();
    let b = t.toLowerCase();
    if (a < b) return -1;
    if (a > b) return 1;
    return 0;
});   // a == ["ant","Bug","cat","Dog"]; case-insensitive sort
```

reverse()

The `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:

```
let a = [1,2,3];
a.reverse();   // a == [3,2,1]
```

7.8.7 Array to String Conversions

The Array class defines three methods that can convert arrays to strings, which is generally something you might do when creating log and error messages. (If you want to save the contents of an array in textual form for later reuse, serialize the array with `JSON.stringify()` [§6.8] instead of using the methods described here.)

The `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:

```
let a = [1, 2, 3];
a.join()              // => "1,2,3"
```

```
a.join(" ")            // => "1 2 3"
a.join("")             // => "123"
let b = new Array(10); // An array of length 10 with no elements
b.join("-")            // => "---------": a string of 9 hyphens
```

The `join()` method is the inverse of the `String.split()` method, which creates an array by breaking a string into pieces.

Arrays, like all JavaScript objects, have a `toString()` method. For an array, this method works just like the `join()` method with no arguments:

```
[1,2,3].toString()          // => "1,2,3"
["a", "b", "c"].toString()  // => "a,b,c"
[1, [2,"c"]].toString()     // => "1,2,c"
```

Note that the output does not include square brackets or any other sort of delimiter around the array value.

`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.

7.8.8 Static Array Functions

In addition to the array methods we've already documented, the Array class also defines three static functions that you can invoke through the `Array` constructor rather than on arrays. `Array.of()` and `Array.from()` are factory methods for creating new arrays. They were documented in §7.1.4 and §7.1.5.

The one other static array function is `Array.isArray()`, which is useful for determining whether an unknown value is an array or not:

```
Array.isArray([])     // => true
Array.isArray({})     // => false
```

7.9 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`.
- `Array.isArray()` returns `true` for arrays.

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:

```
let a = {};  // Start with a regular empty object

// Add properties to make it "array-like"
let i = 0;
while(i < 10) {
    a[i] = i * i;
    i++;
}
a.length = i;

// Now iterate through it as if it were a real array
let total = 0;
for(let j = 0; j < a.length; j++) {
    total += a[j];
}
```

In client-side JavaScript, a number of methods for working with HTML documents (such as `document.querySelectorAll()`, for example) 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
        Number.isFinite(o.length) &&    // o.length is a finite number
        o.length >= 0 &&                // o.length is non-negative
        Number.isInteger(o.length) &&   // o.length is an integer
        o.length < 4294967295) {        // o.length < 2^32 - 1
        return true;                    // Then o is array-like.
    } else {
        return false;                   // Otherwise it is not.
```

```
    }
}
```

We'll see in a later section that strings behave like arrays. Nevertheless, tests like this one for array-like objects typically return `false` for strings—they are usually best handled as strings, not as arrays.

Most 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. 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 (see §8.7.4 for details):

```
let a = {"0": "a", "1": "b", "2": "c", length: 3}; // An array-like object
Array.prototype.join.call(a, "+")                  // => "a+b+c"
Array.prototype.map.call(a, x => x.toUpperCase())  // => ["A","B","C"]
Array.prototype.slice.call(a, 0)   // => ["a","b","c"]: true array copy
Array.from(a)                      // => ["a","b","c"]: easier array copy
```

The second-to-last line of this code invokes the Array `slice()` method on an array-like object in order to copy the elements of that object into a true array object. This is an idiomatic trick that exists in much legacy code, but is now much easier to do with `Array.from()`.

7.10 Strings as Arrays

JavaScript strings behave like read-only arrays of UTF-16 Unicode characters. Instead of accessing individual characters with the `charAt()` method, you can use square brackets:

```
let 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:

```
Array.prototype.join.call("JavaScript", " ")  // => "J a v a S c r i p t"
```

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.

7.11 Summary

This chapter has covered JavaScript arrays in depth, including esoteric details about sparse arrays and array-like objects. The main points to take from this chapter are:

- Array literals are written as comma-separated lists of values within square brackets.
- Individual array elements are accessed by specifying the desired array index within square brackets.
- The `for/of` loop and `...` spread operator introduced in ES6 are particularly useful ways to iterate arrays.
- The Array class defines a rich set of methods for manipulating arrays, and you should be sure to familiarize yourself with the Array API.

CHAPTER 8
Functions

This chapter covers JavaScript functions. Functions are a fundamental building block for JavaScript programs and a common feature in almost all programming languages. You may already be familiar with the concept of a function under a name such as *subroutine* or *procedure*.

A *function* is a block of JavaScript code that is defined once but may be executed, or *invoked*, any number of times. 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 a 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 §6.2 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

The most straightforward way to define a JavaScript function is with the `function` keyword, which can be used as a declaration or as an expression. ES6 defines an important new way to define functions without the `function` keyword: "arrow functions" have a particularly compact syntax and are useful when passing one function as an argument to another function. The subsections that follow cover these three ways of defining functions. Note that some details of function definition syntax involving function parameters are deferred to §8.3.

In object literals and class definitions, there is a convenient shorthand syntax for defining methods. This shorthand syntax was covered in §6.10.5 and is equivalent to using a function definition expression and assigning it to an object property using the basic `name:value` object literal syntax. In another special case, you can use keywords `get` and `set` in object literals to define special property getter and setter methods. This function definition syntax was covered in §6.10.6.

Note that functions can also be defined with the `Function()` constructor, which is the subject of §8.7.7. Also, JavaScript defines some specialized kinds of functions. `function*` defines generator functions (see Chapter 12) and `async function` defines asynchronous functions (see Chapter 13).

8.1.1 Function Declarations

Function declarations consist of the `function` keyword, followed by these components:

- An identifier that names the function. The name is a required part of function declarations: it is used as the name of a variable, and the newly defined function object is assigned to the variable.
- 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.

Here are some example function declarations:

```
// Print the name and value of each property of o.  Return undefined.
function printprops(o) {
    for(let 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) {
    let dx = x2 - x1;
    let 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);
}
```

One of the important things to understand about function declarations is that the name of the function becomes a variable whose value is the function itself. Function declaration statements are "hoisted" to the top of the enclosing script, function, or block so that functions defined in this way may be invoked from code that appears before the definition. Another way to say this is that all of the functions declared in a block of JavaScript code will be defined throughout that block, and they will be defined before the JavaScript interpreter begins to execute any of the code in that block.

The `distance()` and `factorial()` functions we've described are designed to compute a value, and they use `return` to return that value to their caller. 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, the return value of the function is `undefined`.

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`. If a function does not contain a `return` statement, it simply executes each statement in the function body until it reaches the end, and returns the `undefined` value to the caller.

Prior to ES6, function declarations were only allowed at the top level within a JavaScript file or within another function. While some implementations bent the rule, it was not technically legal to define functions inside the body of loops, conditionals, or other blocks. In the strict mode of ES6, however, function declarations are allowed within blocks. A function defined within a block only exists within that block, however, and is not visible outside the block.

8.1.2 Function Expressions

Function expressions look a lot like function declarations, but they appear within the context of a larger expression or statement, and the name is optional. Here are some example function expressions:

```
// This function expression defines a function that squares its argument.
// Note that we assign it to a variable
const square = function(x) { return x*x; };

// Function expressions can include names, which is useful for recursion.
const 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:
[3,2,1].sort(function(a,b) { return a-b; });

// Function expressions are sometimes defined and immediately invoked:
let tensquared = (function(x) {return x*x;}(10));
```

Note that the function name is optional for functions defined as expressions, and most of the preceding function expressions we've shown omit it. A function declaration actually *declares* a variable and assigns a function object to it. A function expression, on the other hand, does not declare a variable: it is up to you to assign the newly defined function object to a constant or variable if you are going to need to refer to it multiple times. It is a good practice to use `const` with function expressions so you don't accidentally overwrite your functions by assigning new values.

A name is allowed for functions, like the factorial function, that need to refer to themselves. If a function 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 (though not nearly as compact as arrow functions, described below).

There is an important difference between defining a function `f()` with a function declaration and assigning a function to the variable `f` after creating it as an expression. When you use the declaration form, the function objects are created before the code that contains them starts to run, and the definitions are hoisted so that you can call these functions from code that appears above the definition statement. This is not true for functions defined as expressions, however: these functions do not exist until the expression that defines them are actually evaluated. Furthermore, 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, so functions defined with expressions cannot be invoked before they are defined.

8.1.3 Arrow Functions

In ES6, you can define functions using a particularly compact syntax known as "arrow functions." This syntax is reminiscent of mathematical notation and uses an `=>` "arrow" to separate the function parameters from the function body. The `function` keyword is not used, and, since arrow functions are expressions instead of statements, there is no need for a function name, either. The general form of an arrow function is a comma-separated list of parameters in parentheses, followed by the `=>` arrow, followed by the function body in curly braces:

```
const sum = (x, y) => { return x + y; };
```

But arrow functions support an even more compact syntax. If the body of the function is a single `return` statement, you can omit the `return` keyword, the semicolon that goes with it, and the curly braces, and write the body of the function as the expression whose value is to be returned:

```
const sum = (x, y) => x + y;
```

Furthermore, if an arrow function has exactly one parameter, you can omit the parentheses around the parameter list:

```
const polynomial = x => x*x + 2*x + 3;
```

Note, however, that an arrow function with no arguments at all must be written with an empty pair of parentheses:

```
const constantFunc = () => 42;
```

Note that, when writing an arrow function, you must not put a new line between the function parameters and the `=>` arrow. Otherwise, you could end up with a line like `const polynomial = x`, which is a syntactically valid assignment statement on its own.

Also, if the body of your arrow function is a single `return` statement but the expression to be returned is an object literal, then you have to put the object literal inside parentheses to avoid syntactic ambiguity between the curly braces of a function body and the curly braces of an object literal:

```
const f = x => { return { value: x }; };  // Good: f() returns an object
const g = x => ({ value: x });            // Good: g() returns an object
const h = x => { value: x };              // Bad: h() returns nothing
const i = x => { v: x, w: x };            // Bad: Syntax Error
```

In the third line of this code, the function `h()` is truly ambiguous: the code you intended as an object literal can be parsed as a labeled statement, so a function that returns undefined is created. On the fourth line, however, the more complicated object literal is not a valid statement, and this illegal code causes a syntax error.

The concise syntax of arrow functions makes them ideal when you need to pass one function to another function, which is a common thing to do with array methods like `map()`, `filter()`, and `reduce()` (see §7.8.1), for example:

```
// Make a copy of an array with null elements removed.
let filtered = [1,null,2,3].filter(x => x !== null); // filtered == [1,2,3]
// Square some numbers:
let squares = [1,2,3,4].map(x => x*x);               // squares == [1,4,9,16]
```

Arrow functions differ from functions defined in other ways in one critical way: they inherit the value of the `this` keyword from the environment in which they are defined rather than defining their own invocation context as functions defined in other ways do. This is an important and very useful feature of arrow functions, and we'll return to it again later in this chapter. Arrow functions also differ from other functions in that they do not have a `prototype` property, which means that they cannot be used as constructor functions for new classes (see §9.2).

8.1.4 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 shown here, 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 will consider them again in §8.6.

8.2 Invoking Functions

The JavaScript code that makes up the body of a function is not executed when the function is defined, but rather when it is invoked. JavaScript functions can be invoked in five ways:

- As functions
- As methods
- As constructors
- Indirectly through their `call()` and `apply()` methods

- Implicitly, via JavaScript language features that do not appear like normal function invocations

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 in the following example. The following code includes a number of regular function invocation expressions:

```
printprops({x: 1});
let total = distance(0,0,2,1) + distance(2,1,3,5);
let 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` statement, then the return value is the value of the expression that follows the `return` or is `undefined` if the `return` statement has no value.

Conditional Invocation

In ES2020 you can insert `?.` after the function expression and before the open parenthesis in a function invocation in order to invoke the function only if it is not `null` or `undefined`. That is, the expression `f?.(x)` is equivalent (assuming no side effects) to:

```
(f !== null && f !== undefined) ? f(x) : undefined
```

Full details on this conditional invocation syntax are in §4.5.1.

For function invocation in non-strict mode, the invocation context (the `this` value) is the global object. In strict mode, however, the invocation context is `undefined`. Note that functions defined using the arrow syntax behave differently: they always inherit the `this` value that is in effect where they are defined.

Functions written to be invoked as functions (and not as methods) do not typically use the `this` keyword at all. The keyword 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.
const strict = (function() { return !this; }());
```

Recursive Functions and the Stack

A *recursive* function is one, like the `factorial()` function at the start of this chapter, that calls itself. Some algorithms, such as those involving tree-based data structures, can be implemented particularly elegantly with recursive functions. When writing a recursive function, however, it is important to think about memory constraints. When a function A calls function B, and then function B calls function C, the JavaScript interpreter needs to keep track of the execution contexts for all three functions. When function C completes, the interpreter needs to know where to resume executing function B, and when function B completes, it needs to know where to resume executing function A. You can imagine these execution contexts as a stack. When a function calls another function, a new execution context is pushed onto the stack. When that function returns, its execution context object is popped off the stack. If a function calls itself recursively 100 times, the stack will have 100 objects pushed onto it, and then have those 100 objects popped off. This call stack takes memory. On modern hardware, it is typically fine to write recursive functions that call themselves hundreds of times. But if a function calls itself ten thousand times, it is likely to fail with an error such as "Maximum call-stack size exceeded."

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 in this example 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, 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 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:

```
let calculator = { // An object literal
    operand1: 1,
    operand2: 1,
    add() {        // We're using method shorthand syntax for this function
        // Note the use of the this keyword to refer to the containing 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") of method invocations as a single expression. When working with Promise-based asynchronous operations (see Chapter 13), for example, it is common to write code structured like this:

```
// Run three asynchronous operations in sequence, handling errors.
doStepOne().then(doStepTwo).then(doStepThree).catch(handleErrors);
```

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*[1] in which an object can be named once and then multiple methods can be invoked on it:

```
new Square().x(100).y(100).size(50).outline("red").fill("blue").draw();
```

Note that `this` is a keyword, not a variable or property name. JavaScript syntax does not allow you to assign a value to `this`.

The `this` keyword is not scoped the way variables are, and, except for arrow functions, nested functions do not inherit the `this` value of the containing function. If a nested function is invoked as a method, its `this` value is the object it was invoked on. If a nested function (that is not an arrow 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 defined within a method and invoked as a function can use `this` to obtain the invocation context of the method. The following code demonstrates the problem:

```
let o = {                 // An object o.
    m: function() {       // Method m of the object.
        let self = this;  // Save the "this" value in a variable.
        this === o        // => true: "this" is the object o.
        f();              // Now call the helper function f().

        function f() {    // A nested function f
            this === o    // => false: "this" is global or undefined
            self === o    // => true: self is the outer "this" value.
        }
    }
};
o.m();                    // Invoke the method m on the object o.
```

1 The term was coined by Martin Fowler. See *http://martinfowler.com/dslCatalog/methodChaining.html*.

Inside the nested function `f()`, the `this` keyword is not equal to the object `o`. This is widely considered to be a flaw in the JavaScript language, and it is important to be aware of it. The code above demonstrates one common workaround. Within the method `m`, we assign the `this` value to a variable `self`, and within the nested function `f`, we can use `self` instead of `this` to refer to the containing object.

In ES6 and later, another workaround to this issue is to convert the nested function `f` into an arrow function, which will properly inherit the `this` value:

```
const f = () => {
    this === o  // true, since arrow functions inherit this
};
```

Functions defined as expressions instead of statements are not hoisted, so in order to make this code work, the function definition for `f` will need to be moved within the method `m` so that it appears before it is invoked.

Another workaround is to invoke the `bind()` method of the nested function to define a new function that is implicitly invoked on a specified object:

```
const f = (function() {
    this === o  // true, since we bound this function to the outer this
}).bind(this);
```

We'll talk more about `bind()` in §8.7.5.

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 §6.2.2, 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. It is not common practice, but you can omit a pair of empty parentheses in a constructor invocation. The following two lines, for example, are equivalent:

```
o = new Object();
o = new Object;
```

A constructor invocation creates a new, empty object that inherits from the object specified by 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 uses 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 §8.7.4.

8.2.5 Implicit Function Invocation

There are various JavaScript language features that do not look like function invocations but that cause functions to be invoked. Be extra careful when writing functions that may be implicitly invoked, because bugs, side effects, and performance issues in these functions are harder to diagnose and fix than in regular functions for the simple reason that it may not be obvious from a simple inspection of your code when they are being called.

The language features that can cause implicit function invocation include:

- If an object has getters or setters defined, then querying or setting the value of its properties may invoke those methods. See §6.10.6 for more information.
- When an object is used in a string context (such as when it is concatenated with a string), its `toString()` method is called. Similarly, when an object is used in a numeric context, its `valueOf()` method is invoked. See §3.9.3 for details.
- When you loop over the elements of an iterable object, there are a number of method calls that occur. Chapter 12 explains how iterators work at the function call level and demonstrates how to write these methods so that you can define your own iterable types.

- A tagged template literal is a function invocation in disguise. §14.5 demonstrates how to write functions that can be used in conjunction with template literal strings.
- Proxy objects (described in §14.7) have their behavior completely controlled by functions. Just about any operation on one of these objects will cause a function to be invoked.

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 and Defaults

When a function is invoked with fewer arguments than declared parameters, the additional parameters are set to their default value, which is normally `undefined`. It is often useful to write functions so that some arguments are optional. Following 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, a) {
    if (a === undefined) a = [];  // If undefined, use a new array
    for(let property in o) a.push(property);
    return a;
}

// getPropertyNames() can be invoked with one or two arguments:
let o = {x: 1}, p = {y: 2, z: 3};  // Two objects for testing
let a = getPropertyNames(o); // a == ["x"]; get o's properties in a new array
getPropertyNames(p, a);      // a == ["x","y","z"]; add p's properties to it
```

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: they would have to explicitly pass `undefined` as the first argument.

In ES6 and later, you can define a default value for each of your function parameters directly in the parameter list of your function. Simply follow the parameter name with an equals sign and the default value to use when no argument is supplied for that parameter:

```
// 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, a = []) {
    for(let property in o) a.push(property);
    return a;
}
```

Parameter default expressions are evaluated when your function is called, not when it is defined, so each time this `getPropertyNames()` function is invoked with one argument, a new empty array is created and passed.[2] It is probably easiest to reason about functions if the parameter defaults are constants (or literal expressions like `[]` and `{}`). But this is not required: you can use variables, or function invocations, for example, to compute the default value of a parameter. One interesting case is that, for functions with multiple parameters, you can use the value of a previous parameter to define the default value of the parameters that follow it:

```
// This function returns an object representing a rectangle's dimensions.
// If only width is supplied, make it twice as high as it is wide.
const rectangle = (width, height=width*2) => ({width, height});
rectangle(1)  // => { width: 1, height: 2 }
```

This code demonstrates that parameter defaults work with arrow functions. The same is true for method shorthand functions and all other forms of function definitions.

8.3.2 Rest Parameters and Variable-Length Argument Lists

Parameter defaults enable us to write functions that can be invoked with fewer arguments than parameters. *Rest parameters* enable the opposite case: they allow us to write functions that can be invoked with arbitrarily more arguments than parameters. Here is an example function that expects one or more numeric arguments and returns the largest one:

2 If you are familiar with Python, note that this is different than Python, in which every invocation shares the same default value.

```
function max(first=-Infinity, ...rest) {
    let maxValue = first; // Start by assuming the first arg is biggest
    // Then loop through the rest of the arguments, looking for bigger
    for(let n of rest) {
        if (n > maxValue) {
            maxValue = n;
        }
    }
    // Return the biggest
    return maxValue;
}

max(1, 10, 100, 2, 3, 1000, 4, 5, 6)  // => 1000
```

A rest parameter is preceded by three periods, and it must be the last parameter in a function declaration. When you invoke a function with a rest parameter, the arguments you pass are first assigned to the non-rest parameters, and then any remaining arguments (i.e., the "rest" of the arguments) are stored in an array that becomes the value of the rest parameter. This last point is important: within the body of a function, the value of a rest parameter will always be an array. The array may be empty, but a rest parameter will never be `undefined`. (It follows from this that it is never useful—and not legal—to define a parameter default for a rest parameter.)

Functions like the previous example that can accept any number of arguments are called *variadic functions*, *variable arity functions*, or *vararg functions*. This book uses the most colloquial term, *varargs*, which dates to the early days of the C programming language.

Don't confuse the `...` that defines a rest parameter in a function definition with the `...` spread operator, described in §8.3.4, which can be used in function invocations.

8.3.3 The Arguments Object

Rest parameters were introduced into JavaScript in ES6. Before that version of the language, varargs functions were written using the Arguments object: within the body of any function, the identifier `arguments` refers to the Arguments object for that invocation. The Arguments object is an array-like object (see §7.9) that allows the argument values passed to the function to be retrieved by number, rather than by name. Here is the `max()` function from earlier, rewritten to use the Arguments object instead of a rest parameter:

```
function max(x) {
    let maxValue = -Infinity;
    // Loop through the arguments, looking for, and remembering, the biggest.
    for(let i = 0; i < arguments.length; i++) {
        if (arguments[i] > maxValue) maxValue = arguments[i];
    }
```

```
    // Return the biggest
    return maxValue;
}

max(1, 10, 100, 2, 3, 1000, 4, 5, 6)  // => 1000
```

The Arguments object dates back to the earliest days of JavaScript and carries with it some strange historical baggage that makes it inefficient and hard to optimize, especially outside of strict mode. You may still encounter code that uses the Arguments object, but you should avoid using it in any new code you write. When refactoring old code, if you encounter a function that uses `arguments`, you can often replace it with a `...args` rest parameter. Part of the unfortunate legacy of the Arguments object is that, in strict mode, `arguments` is treated as a reserved word, and you cannot declare a function parameter or a local variable with that name.

8.3.4 The Spread Operator for Function Calls

The spread operator `...` is used to unpack, or "spread out," the elements of an array (or any other iterable object, such as strings) in a context where individual values are expected. We've seen the spread operator used with array literals in §7.1.2. The operator can be used, in the same way, in function invocations:

```
let numbers = [5, 2, 10, -1, 9, 100, 1];
Math.min(...numbers)  // => -1
```

Note that `...` is not a true operator in the sense that it cannot be evaluated to produce a value. Instead, it is a special JavaScript syntax that can be used in array literals and function invocations.

When we use the same `...` syntax in a function definition rather than a function invocation, it has the opposite effect to the spread operator. As we saw in §8.3.2, using `...` in a function definition gathers multiple function arguments into an array. Rest parameters and the spread operator are often useful together, as in the following function, which takes a function argument and returns an instrumented version of the function for testing:

```
// This function takes a function and returns a wrapped version
function timed(f) {
    return function(...args) {  // Collect args into a rest parameter array
        console.log(`Entering function ${f.name}`);
        let startTime = Date.now();
        try {
            // Pass all of our arguments to the wrapped function
            return f(...args);  // Spread the args back out again
        }
        finally {
            // Before we return the wrapped return value, print elapsed time.
            console.log(`Exiting ${f.name} after ${Date.now()-startTime}ms`);
        }
```

```
    };
}

// Compute the sum of the numbers between 1 and n by brute force
function benchmark(n) {
    let sum = 0;
    for(let i = 1; i <= n; i++) sum += i;
    return sum;
}

// Now invoke the timed version of that test function
timed(benchmark)(1000000) // => 500000500000; this is the sum of the numbers
```

8.3.5 Destructuring Function Arguments into Parameters

When you invoke a function with a list of argument values, those values end up being assigned to the parameters declared in the function definition. This initial phase of function invocation is a lot like variable assignment. So it should not be surprising that we can use the techniques of destructuring assignment (see §3.10.3) with functions.

If you define a function that has parameter names within square brackets, you are telling the function to expect an array value to be passed for each pair of square brackets. As part of the invocation process, the array arguments will be unpacked into the individually named parameters. As an example, suppose we are representing 2D vectors as arrays of two numbers, where the first element is the X coordinate and the second element is the Y coordinate. With this simple data structure, we could write the following function to add two vectors:

```
function vectorAdd(v1, v2) {
    return [v1[0] + v2[0], v1[1] + v2[1]];
}
vectorAdd([1,2], [3,4])  // => [4,6]
```

The code would be easier to understand if we destructured the two vector arguments into more clearly named parameters:

```
function vectorAdd([x1,y1], [x2,y2]) { // Unpack 2 arguments into 4 parameters
    return [x1 + x2, y1 + y2];
}
vectorAdd([1,2], [3,4])  // => [4,6]
```

Similarly, if you are defining a function that expects an object argument, you can destructure parameters of that object. Let's use a vector example again, except this time, let's suppose that we represent vectors as objects with `x` and `y` parameters:

```
// Multiply the vector {x,y} by a scalar value
function vectorMultiply({x, y}, scalar) {
    return { x: x*scalar, y: y*scalar };
}
vectorMultiply({x: 1, y: 2}, 2)  // => {x: 2, y: 4}
```

This example of destructuring a single object argument into two parameters is a fairly clear one because the parameter names we use match the property names of the incoming object. The syntax is more verbose and more confusing when you need to destructure properties with one name into parameters with different names. Here's the vector addition example, implemented for object-based vectors:

```
function vectorAdd(
    {x: x1, y: y1}, // Unpack 1st object into x1 and y1 params
    {x: x2, y: y2}  // Unpack 2nd object into x2 and y2 params
)
{
    return { x: x1 + x2, y: y1 + y2 };
}
vectorAdd({x: 1, y: 2}, {x: 3, y: 4})  // => {x: 4, y: 6}
```

The tricky thing about destructuring syntax like `{x:x1, y:y1}` is remembering which are the property names and which are the parameter names. The rule to keep in mind for destructuring assignment and destructuring function calls is that the variables or parameters being declared go in the spots where you'd expect values to go in an object literal. So property names are always on the lefthand side of the colon, and the parameter (or variable) names are on the right.

You can define parameter defaults with destructured parameters. Here's vector multiplication that works with 2D or 3D vectors:

```
// Multiply the vector {x,y} or {x,y,z} by a scalar value
function vectorMultiply({x, y, z=0}, scalar) {
    return { x: x*scalar, y: y*scalar, z: z*scalar };
}
vectorMultiply({x: 1, y: 2}, 2)  // => {x: 2, y: 4, z: 0}
```

Some languages (like Python) allow the caller of a function to invoke a function with arguments specified in `name=value` form, which is convenient when there are many optional arguments or when the parameter list is long enough that it is hard to remember the correct order. JavaScript does not allow this directly, but you can approximate it by destructuring an object argument into your function parameters. Consider a function that copies a specified number of elements from one array into another array with optionally specified starting offsets for each array. Since there are five possible parameters, some of which have defaults, and it would be hard for a caller to remember which order to pass the arguments in, we can define and invoke the `arraycopy()` function like this:

```
function arraycopy({from, to=from, n=from.length, fromIndex=0, toIndex=0}) {
    let valuesToCopy = from.slice(fromIndex, fromIndex + n);
    to.splice(toIndex, 0, ...valuesToCopy);
    return to;
}
let a = [1,2,3,4,5], b = [9,8,7,6,5];
arraycopy({from: a, n: 3, to: b, toIndex: 4}) // => [9,8,7,6,1,2,3,5]
```

When you destructure an array, you can define a rest parameter for extra values within the array that is being unpacked. That rest parameter within the square brackets is completely different than the true rest parameter for the function:

```
// This function expects an array argument. The first two elements of that
// array are unpacked into the x and y parameters. Any remaining elements
// are stored in the coords array. And any arguments after the first array
// are packed into the rest array.
function f([x, y, ...coords], ...rest) {
    return [x+y, ...rest, ...coords];  // Note: spread operator here
}
f([1, 2, 3, 4], 5, 6)   // => [3, 5, 6, 3, 4]
```

In ES2018, you can also use a rest parameter when you destructure an object. The value of that rest parameter will be an object that has any properties that did not get destructured. Object rest parameters are often useful with the object spread operator, which is also a new feature of ES2018:

```
// Multiply the vector {x,y} or {x,y,z} by a scalar value, retain other props
function vectorMultiply({x, y, z=0, ...props}, scalar) {
    return { x: x*scalar, y: y*scalar, z: z*scalar, ...props };
}
vectorMultiply({x: 1, y: 2, w: -1}, 2)  // => {x: 2, y: 4, z: 0, w: -1}
```

Finally, keep in mind that, in addition to destructuring argument objects and arrays, you can also destructure arrays of objects, objects that have array properties, and objects that have object properties, to essentially any depth. Consider graphics code that represents circles as objects with `x`, `y`, `radius`, and `color` properties, where the `color` property is an array of red, green, and blue color components. You might define a function that expects a single circle object to be passed to it but destructures that circle object into six separate parameters:

```
function drawCircle({x, y, radius, color: [r, g, b]}) {
    // Not yet implemented
}
```

If function argument destructuring is any more complicated than this, I find that the code becomes harder to read, rather than simpler. Sometimes, it is clearer to be explicit about your object property access and array indexing.

8.3.6 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 make your code self-documenting by choosing descriptive names for function arguments and by documenting them carefully in the comments for each function. (Alternatively, see §17.8 for a language extension that allows you to layer type checking on top of regular JavaScript.)

As described in §3.9, 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 one or two array arguments and will fail if these arguments are of the wrong type. Unless you are writing a private function that will only be called from nearby parts of your code, 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:

```
// Return the sum of the elements an iterable object a.
// The elements of a must all be numbers.
function sum(a) {
    let total = 0;
    for(let element of a) { // Throws TypeError if a is not iterable
        if (typeof element !== "number") {
            throw new TypeError("sum(): elements must be numbers");
        }
        total += element;
    }
    return total;
}
sum([1,2,3])    // => 6
sum(1, 2, 3);   // !TypeError: 1 is not iterable
sum([1,2,"3"]); // !TypeError: element 2 is not a number
```

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; }
```

3 This may not seem like a particularly interesting point unless you are familiar with more static languages, in which functions are part of a program but cannot be manipulated by the program.

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:

```
let 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. As we've already discussed, we call the functions "methods" when we do this:

```
let o = {square: function(x) { return x*x; }}; // An object literal
let y = o.square(16);                          // y == 256
```

Functions don't even require names at all, as when they're assigned to array elements:

```
let a = [x => 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!

As an example of how useful it is to treat 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 §7.8.6.

Example 8-1 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 8-1. 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 preceding 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):
let 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 within an object literal;
const operators = {
    add:      (x,y) => x+y,
    subtract: (x,y) => x-y,
    multiply: (x,y) => x*y,
    divide:   (x,y) => x/y,
    pow:      Math.pow  // This 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";
}

operate2("add", "hello", operate2("add", " ", "world")) // => "hello world"
operate2("pow", 10, 2)  // => 100
```

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 itself. 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++;  // Return and increment counter property
```

```
}
uniqueInteger()  // => 0
uniqueInteger()  // => 1
```

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 (Number.isInteger(n) && n > 0) {           // Positive integers 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.
factorial(6)  // => 720
factorial[5]  // => 120; the call above caches this value
```

8.5 Functions as Namespaces

Variables declared within a function are not visible outside of the function. 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 cluttering the global namespace.

Suppose, for example, you have a chunk 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 chunk of code will be used in many different programs, you don't know whether the variables it creates will conflict with variables created by the programs that use it. The solution is to put the chunk of code into a function and then invoke the function. This way, variables that would have been global become local to the function:

```
function chunkNamespace() {
    // Chunk of code goes here
    // Any variables defined in the chunk are local to this function
    // instead of cluttering up the global namespace.
}
chunkNamespace();  // But don't forget to invoke the function!
```

This code defines only a single global variable: the function name `chunkNamespace`. If defining even a single property is too much, you can define and invoke an anonymous function in a single expression:

```
(function() {  // chunkNamespace() function rewritten as an unnamed expression.
    // Chunk of 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 and has been given the name "immediately invoked function expression." Note the use of parentheses in the previous code example. 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. The leading parenthesis also helps human readers recognize when a function is being defined to be immediately invoked instead of defined for later use.

This use of functions as namespaces becomes really useful when we define one or more functions inside the namespace function using variables within that namesapce, but then pass them back out as the return value of the namespace function. Functions like this are known as *closures*, and they're the topic of the next section.

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 scope in which the function definition appears. 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.

Technically, all JavaScript functions are closures, but because most functions are invoked from the same scope that they were defined in, it normally doesn't really matter that there is a closure involved. Closures become interesting when they are invoked from a different scope than the one they were defined in. 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:

```
let scope = "global scope";          // A global variable
function checkscope() {
```

```
    let 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?

```
let scope = "global scope";          // A global variable
function checkscope() {
    let scope = "local scope";       // A local variable
    function f() { return scope; }   // Return the value in scope here
    return f;
}
let s = 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 they were defined in. The nested function `f()` was defined in a scope where the variable `scope` was bound to the value "local scope". That binding is still in effect when `f` is executed, no matter where it is executed from. So the last line of the preceding code example 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.

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()` using an immediately invoked function expression to define a namespace and a closure that uses that namespace to keep its state private:

```
let uniqueInteger = (function() {  // Define and invoke
    let counter = 0;               // Private state of function below
    return function() { return counter++; };
}());
uniqueInteger()  // => 0
uniqueInteger()  // => 1
```

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 its 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. Consider the following code:

```
function counter() {
    let n = 0;
    return {
        count: function() { return n++; },
        reset: function() { n = 0; }
    };
}

let 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—independent of the scopes used by previous invocations—and a new private variable within that scope. 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 §6.10.6, 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");
        }
    };
}

let c = counter(1000);
c.count            // => 1000
c.count            // => 1001
c.count = 2000;
c.count            // => 2000
c.count = 2000;    // !Error: count can only be set to a larger value
```

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 8-2 is a generalization of the shared private state through the 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 8-2. 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) {
    let 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 new TypeError(`set${name}: invalid value ${v}`);
        } else {
            value = v;
        }
    };
}

// The following code demonstrates the addPrivateProperty() method.
let o = {};  // Here is an empty object

// Add property accessor methods getName and setName()
// Ensure that only string values are allowed
addPrivateProperty(o, "Name", x => typeof x === "string");

o.setName("Frank");       // Set the property value
o.getName()               // => "Frank"
o.setName(0);             // !TypeError: 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 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 () => v; }

// Create an array of constant functions:
let 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() {
    let funcs = [];
    for(var i = 0; i < 10; i++) {
        funcs[i] = () => i;
    }
    return funcs;
}

let funcs = constfuncs();
funcs[5]()    // => 10; Why doesn't this return 5?
```

This code 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 associated with a closure is "live." Nested functions do not make private copies of the scope or make static snapshots of the variable bindings. Fundamentally, the problem here is that variables declared with `var` are defined throughout the function. Our `for` loop declares the loop variable with `var i`, so the variable `i` is defined throughout the function rather than being more narrowly scoped to the body of the loop. The code demonstrates a common category of bugs in ES5 and before, but the introduction of block-scoped variables in ES6 addresses the issue. If we just replace the `var` with a `let` or a `const`, then the problem goes away. Because `let` and `const` are block scoped, each iteration of the loop defines a scope that is independent of the scopes for all other iterations, and each of these scopes has its own independent binding of `i`.

Another thing to remember when writing closures is that `this` is a JavaScript keyword, not a variable. As discussed earlier, arrow functions inherit the `this` value of the function that contains them, but functions defined with the `function` keyword do not. So if you're writing a closure that needs to use the `this` value of its containing function, you should use an arrow function, or call `bind()`, on the closure before returning it, or assign the outer `this` value to a variable that your closure will inherit:

```
const self = this;  // Make the this value available to nested functions
```

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 the `length`, `name`, and `prototype` properties; the `call()`, `apply()`, `bind()`, and `toString()` methods; and the `Function()` constructor.

8.7.1 The length Property

The read-only `length` property of a function specifies 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. If a function has a rest parameter, that parameter is not counted for the purposes of this `length` property.

8.7.2 The name Property

The read-only `name` property of a function specifies the name that was used when the function was defined, if it was defined with a name, or the name of the variable or property that an unnamed function expression was assigned to when it was first created. This property is primarily useful when writing debugging or error messages.

8.7.3 The prototype Property

All functions, except arrow functions, have 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 §6.2.3 and will be covered again in Chapter 9.

8.7.4 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. 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 these lines of code 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.
```

Remember that arrow functions inherit the `this` value of the context where they are defined. This cannot be overridden with the `call()` and `apply()` methods. If you call either of those methods on an arrow function, the first argument is effectively ignored.

Any arguments to `call()` after the first invocation context argument are the values that are passed to the function that is invoked (and these arguments are not ignored for arrow functions). 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. In ES6 and later, we can just use the spread operator, but you may see ES5 code that uses `apply()` instead. For example, to find the largest number in an array of numbers without using the spread operator, you could use the `apply()` method to pass the elements of the array to the `Math.max()` function:

```
let biggest = Math.max.apply(Math, arrayOfNumbers);
```

The `trace()` function defined in the following is similar to the `timed()` function defined in §8.3.4, but it works for methods instead of functions. It uses the `apply()` method instead of a spread operator, and by doing that, it is able to invoke the wrapped method with the same arguments and the same `this` value as the wrapper method:

```
// 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) {
    let original = o[m];         // Remember original method in the closure.
    o[m] = function(...args) {   // Now define the new method.
        console.log(new Date(), "Entering:", m);      // Log message.
        let result = original.apply(this, args);      // Invoke original.
        console.log(new Date(), "Exiting:", m);       // Log message.
        return result;                                // Return result.
    };
}
```

8.7.5 The bind() Method

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
let o = { x: 1 };                    // An object we'll bind to
let g = f.bind(o);                   // Calling g(x) invokes f() on o
g(2)                                 // => 3
let p = { x: 10, g };                // Invoke g() as a method of this object
p.g(2)                               // => 3: g is still bound to o, not p.
```

Arrow functions inherit their `this` value from the environment in which they are defined, and that value cannot be overridden with `bind()`, so if the function `f()` in

the preceding code was defined as an arrow function, the binding would not work. The most common use case for calling `bind()` is to make non-arrow functions behave like arrow functions, however, so this limitation on binding arrow functions is not a problem in practice.

The `bind()` method does more than just bind a function to an object, however. It can also perform partial application: any arguments you pass to `bind()` after the first are bound along with the `this` value. This partial application feature of `bind()` does work with arrow functions. 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:

```
let sum = (x,y) => x + y;      // Return the sum of 2 args
let succ = sum.bind(null, 1);  // Bind the first argument to 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; }
let 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
```

The `name` property of the function returned by `bind()` is the name property of the function that `bind()` was called on, prefixed with the word "bound".

8.7.6 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.

8.7.7 The Function() Constructor

Because functions are objects, there is a `Function()` constructor that can be used to create new functions:

```
const 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:

```
const 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 parameter names for the function. If you are defining a

function that takes no arguments, you would 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 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:

```
let scope = "global";
function constructFunction() {
    let scope = "local";
    return new Function("return scope");  // Doesn't capture 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 will probably never need to use this constructor in your code.

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. 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.

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:

```
let 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
let total = 0;
for(let i = 0; i < data.length; i++) total += data[i];
let mean = total/data.length;  // mean == 3; 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(let i = 0; i < data.length; i++) {
    let deviation = data[i] - mean;
    total += deviation * deviation;
}
let stddev = Math.sqrt(total/(data.length-1));  // stddev == 2
```

We can perform these same computations in concise functional style using the array methods `map()` and `reduce()` like this (see §7.8.1 to review these methods):

```
// First, define two simple functions
const sum = (x,y) => x+y;
const square = x => x*x;

// Then use those functions with Array methods to compute mean and stddev
let data = [1,1,3,5,5];
let mean = data.reduce(sum)/data.length;  // mean == 3
let deviations = data.map(x => x-mean);
let stddev = Math.sqrt(deviations.map(square).reduce(sum)/(data.length-1));
stddev  // => 2
```

This new version of the code looks quite different than the first one, but it is still invoking methods on objects, so it has some object-oriented conventions remaining. Let's write functional versions of the `map()` and `reduce()` methods:

```
const map = function(a, ...args) { return a.map(...args); };
const reduce = function(a, ...args) { return a.reduce(...args); };
```

With these `map()` and `reduce()` functions defined, our code to compute the mean and standard deviation now looks like this:

```
const sum = (x,y) => x+y;
const square = x => x*x;

let data = [1,1,3,5,5];
let mean = reduce(data, sum)/data.length;
let deviations = map(data, x => x-mean);
let stddev = Math.sqrt(reduce(map(deviations, square), sum)/(data.length-1));
stddev  // => 2
```

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(...args) {            // Return a new function
        let result = f.apply(this, args); // that calls f
        return !result;                   // and negates its result.
    };
}

const even = x => x % 2 === 0; // A function to determine if a number is even
const 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
```

This `not()` function is a higher-order function because it takes a function argument and returns a new function. As another example, consider the `mapper()` function that follows. 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 a => map(a, f);
}

const increment = x => x+1;
const incrementAll = mapper(increment);
incrementAll([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, then passes
// the return value of g to f, 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(...args) {
        // 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, args));
    };
}

const sum = (x,y) => x+y;
```

```
const square = x => x*x;
compose(square, sum)(2,3)  // => 25; the square of the sum
```

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` (see §8.7.5) 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:

```
// The arguments to this function are passed on the left
function partialLeft(f, ...outerArgs) {
    return function(...innerArgs) { // Return this function
        let args = [...outerArgs, ...innerArgs]; // Build the argument list
        return f.apply(this, args);              // Then invoke f with it
    };
}

// The arguments to this function are passed on the right
function partialRight(f, ...outerArgs) {
    return function(...innerArgs) {  // Return this function
        let args = [...innerArgs, ...outerArgs]; // Build the argument list
        return f.apply(this, args);              // Then invoke f with it
    };
}

// 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, ...outerArgs) {
    return function(...innerArgs) {
        let args = [...outerArgs]; // local copy of outer args template
        let innerIndex=0;          // which inner arg is next
        // Loop through the args, filling in undefined values from inner args
        for(let i = 0; i < args.length; i++) {
            if (args[i] === undefined) args[i] = innerArgs[innerIndex++];
        }
        // Now append any remaining inner arguments
        args.push(...innerArgs.slice(innerIndex));
        return f.apply(this, args);
    };
}

// Here is a function with three arguments
const 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:

```
const increment = partialLeft(sum, 1);
const cuberoot = partialRight(Math.pow, 1/3);
cuberoot(increment(26))  // => 3
```

Partial application becomes even more interesting when we combine it with other higher-order functions. Here, for example, is a way to define the preceding `not()` function just shown using composition and partial application:

```
const not = partialLeft(compose, x => !x);
const even = x => x % 2 === 0;
const odd = not(even);
const isNumber = not(isNaN);
odd(3) && isNumber(2)  // => true
```

We can also use composition and partial application to redo our mean and standard deviation calculations in extreme functional style:

```
// sum() and square() functions are defined above. Here are some more:
const product = (x,y) => x*y;
const neg = partial(product, -1);
const sqrt = partial(Math.pow, undefined, .5);
const reciprocal = partial(Math.pow, undefined, neg(1));

// Now compute the mean and standard deviation.
let data = [1,1,3,5,5];   // Our data
let mean = product(reduce(data, sum), reciprocal(data.length));
let stddev = sqrt(product(reduce(map(data,
                                     compose(square,
                                             partial(sum, neg(mean)))),
                                 sum),
                          reciprocal(sum(data.length,neg(1)))));
[mean, stddev]  // => [3, 2]
```

Notice that this code to compute mean and standard deviation is entirely function invocations; there are no operators involved, and the number of parentheses has grown so large that this JavaScript is beginning to look like Lisp code. Again, this is not a style that I advocate for JavaScript programming, but it is an interesting exercise to see how deeply functional JavaScript code can be.

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 that follows 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) {
    const cache = new Map();  // Value cache stored in the closure.

    return function(...args) {
        // Create a string version of the arguments to use as a cache key.
        let key = args.length + args.join("+");
        if (cache.has(key)) {
            return cache.get(key);
        } else {
            let result = f.apply(this, args);
            cache.set(key, result);
            return result;
        }
    };
}
```

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
    if (a < b) {           // Ensure that a >= b when we start
        [a, b] = [b, a];   // Destructuring assignment to swap variables
    }
    while(b !== 0) {       // This is Euclid's algorithm for GCD
        [a, b] = [b, a%b];
    }
    return a;
}

const 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.
const factorial = memoize(function(n) {
    return (n <= 1) ? 1 : n * factorial(n-1);
});
factorial(5)      // => 120: also caches values for 4, 3, 2 and 1.
```

8.9 Summary

Some key points to remember about this chapter are as follows:

- You can define functions with the `function` keyword and with the ES6 `=>` arrow syntax.
- You can invoke functions, which can be used as methods and constructors.
- Some ES6 features allow you to define default values for optional function parameters, to gather multiple arguments into an array using a rest parameter, and to destructure object and array arguments into function parameters.
- You can use the `...` spread operator to pass the elements of an array or other iterable object as arguments in a function invocation.
- A function defined inside of and returned by an enclosing function retains access to its lexical scope and can therefore read and write the variables defined inside the outer function. Functions used in this way are called *closures*, and this is a technique that is worth understanding.
- Functions are objects that can be manipulated by JavaScript, and this enables a functional style of programming.

CHAPTER 9
Classes

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 methods that define their behavior. These methods are defined by the class and shared by all instances. Imagine a class named Complex that represents and performs arithmetic on complex numbers, for example. A Complex instance would have properties to hold the real and imaginary parts (the state) of the complex number. And the Complex class would define methods to perform addition and multiplication (the behavior) of those numbers.

In JavaScript, classes use prototype-based inheritance: if two objects inherit properties (generally function-valued properties, or methods) from the same prototype, then we say that those objects are instances of the same class. That, in a nutshell, is how JavaScript classes work. JavaScript prototypes and inheritance were covered in §6.2.3 and §6.3.2, and you will need to 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 or factory function. Constructors have been covered in §4.6, §6.2.2, and §8.2.3, and this chapter has more in §9.2.

JavaScript has always allowed the definition of classes. ES6 introduced a brand-new syntax (including a `class` keyword) that makes it even easier to create classes. These new JavaScript classes work in the same way that old-style classes do, and this chapter starts by explaining the old way of creating classes because that demonstrates more clearly what is going on behind the scenes to make classes work. Once we've

explained those fundamentals, we'll shift and start using the new, simplified class definition syntax.

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.

9.1 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. Chapter 6 covered the `Object.create()` function that returns a newly created object that inherits from a specified prototype object. If we define a prototype object and then use `Object.create()` 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

```
// This is a factory function that returns a new range object.
function range(from, to) {
    // Use Object.create() 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.
    let r = Object.create(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(x) { return this.from <= x && x <= this.to; },

    // A generator function that makes instances of the class iterable.
    // Note that it only works for numeric ranges.
    *[Symbol.iterator]() {
        for(let x = Math.ceil(this.from); x <= this.to; x++) yield x;
    },

    // Return a string representation of the range
    toString() { return "(" + this.from + "..." + this.to + ")"; }
};

// Here are example uses of a range object.
let r = range(1,3);      // Create a range object
r.includes(2)            // => true: 2 is in the range
r.toString()             // => "(1...3)"
[...r]                   // => [1, 2, 3]; convert to an array via iterator
```

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.
- It uses the `methods` property of this `range()` function 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.
- 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.
- The `range.methods` object uses the ES6 shorthand syntax for defining methods, which is why you don't see the `function` keyword anywhere. (See §6.10.5 to review object literal shorthand method syntax.)
- One of the methods in the prototype has the computed name (§6.10.2) `Symbol.iterator`, which means that it is defining an iterator for Range objects. The name of this method is prefixed with `*`, which indicates that it is a generator function instead of a regular function. Iterators and generators are covered in detail in Chapter 12. For now, the upshot is that instances of this Range class can be used with the `for/of` loop and with the `...` spread operator.
- The shared, inherited methods defined in `range.methods` all use the `from` and `to` properties that were initialized in the `range()` factory function. 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 a simple 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. §6.2.3 introduced prototypes and emphasized that while almost all objects have a prototype, only a few objects have a `prototype` property. Finally, we can clarify this: it is function objects that have a `prototype` property. This means that all objects created with the same constructor function 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 demonstrates the idiomatic way to create a class in versions of JavaScript that do not support the ES6 `class` keyword. Even though `class` is well supported now, there is still lots of older JavaScript code around that defines classes like this, and you should be familiar with the idiom so that you can read old code and so that you understand what is going on "under the hood" when you use the `class` keyword.

Example 9-2. A Range class using a constructor

```
// 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; },

    // A generator function that makes instances of the class iterable.
    // Note that it only works for numeric ranges.
    [Symbol.iterator]: function*() {
        for(let x = Math.ceil(this.from); x <= this.to; x++) yield x;
    },
```

```
    // Return a string representation of the range
    toString: function() { return "(" + this.from + "..." + this.to + ")"; }
};

// Here are example uses of this new Range class
let r = new Range(1,3);   // Create a Range object; note the use of new
r.includes(2)             // => true: 2 is in the range
r.toString()              // => "(1...3)"
[...r]                    // => [1, 2, 3]; convert to an array via iterator
```

It is worth comparing Examples 9-1 and 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 (by convention) 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 `Object.create()` 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 know when to use `new`.

Constructors and new.target

Within a function body, you can tell whether the function has been invoked as a constructor with the special expression `new.target`. If the value of that expression is defined, then you know that the function was invoked as a constructor, with the `new` keyword. When we discuss subclasses in §9.5, we'll see that `new.target` is not always a reference to the constructor it is used in: it might also refer to the constructor function of a subclass.

If `new.target` is `undefined`, then the containing function was invoked as a function, without the `new` keyword. JavaScript's various error constructors can be invoked

without `new`, and if you want to emulate this feature in your own constructors, you can write them like this:

```
function C() {
    if (!new.target) return new C();
    // initialization code goes here
}
```

This technique only works for constructors defined in this old-fashioned way. Classes created with the `class` keyword do not allow their constructors to be invoked without `new`.

Another critical difference between Examples 9-1 and 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 Examples 9-1 and 9-2 : the range methods are defined and invoked in the same way for both classes. Because Example 9-2 demonstrates the idiomatic way to create classes in versions of JavaScript before ES6, it does not use the ES6 shorthand method syntax in the prototype object and explicitly spells out the methods with the `function` keyword. But you can see that the implementation of the methods is the same in both examples.

Importantly, note that neither of the two range examples uses arrow functions when defining constructors or methods. Recall from §8.1.3 that functions defined in this way do not have a `prototype` property and so cannot be used as constructors. Also, arrow functions inherit the `this` keyword from the context in which they are defined rather than setting it based on the object through which they are invoked, and this makes them useless for methods because the defining characteristic of methods is that they use `this` to refer to the instance on which they were invoked.

Fortunately, the new ES6 class syntax doesn't allow the option of defining methods with arrow functions, so this is not a mistake that you can accidentally make when using that syntax. We will cover the ES6 `class` keyword soon, but first, there are more details to cover about constructors.

9.2.1 Constructors, Class Identity, and instanceof

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 though 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 as the righthand operand of 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   // => true: r inherits from Range.prototype
```

The `instanceof` operator was described in §4.9.4. The lefthand operand should be the object that is being tested, and the righthand 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`.

Technically speaking, in the previous code example, the `instanceof` operator is not checking whether `r` was actually initialized by the `Range` constructor. Instead, it is checking whether `r` inherits from `Range.prototype`. If we define a function `Strange()` and set its prototype to be the same as `Range.prototype`, then objects created with `new Strange()` will count as Range objects as far as `instanceof` is concerned (they won't actually work as Range objects, however, because their `from` and `to` properties have not been initialized):

```
function Strange() {}
Strange.prototype = Range.prototype;
new Strange() instanceof Range   // => true
```

Even though `instanceof` cannot actually verify the use of a constructor, it still uses a constructor function as its righthand side because constructors are the public identity of a class.

If you want to test the prototype chain of an object for a specific prototype and do not want to use the constructor function as an intermediary, you can use the `isPrototypeOf()` method. In Example 9-1, for example, we defined a class without a constructor function, so there is no way to use `instanceof` with that class. Instead, however, we could test whether an object `r` was a member of that constructor-less class with this code:

```
range.methods.isPrototypeOf(r);  // range.methods is the prototype object.
```

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 regular JavaScript function (excluding arrow functions, generator functions, and async functions) can be used as a constructor, and constructor invocations need a `prototype` property. Therefore, every regular JavaScript function[1] automatically has a `prototype` property. The value of this property is an object that has a single, non-enumerable `constructor` property. The value of the `constructor` property is the function object:

```
let F = function() {}; // This is a function object.
let p = F.prototype;   // This is the prototype object associated with F.
let c = p.constructor; // This is the function associated with the prototype.
c === F                // => true: F.prototype.constructor === F for any F
```

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:

```
let 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.

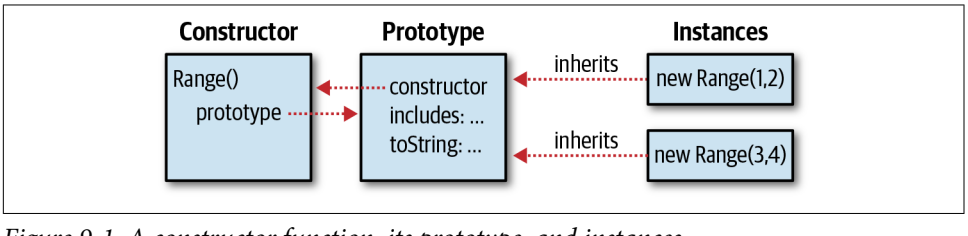

Figure 9-1. A constructor function, its prototype, and instances

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:

1 Except functions returned by the ES5 `Function.bind()` method. Bound functions have no prototype property of their own, but they use the prototype of the underlying function if they are invoked as constructors.

```
Range.prototype = {
    constructor: Range,  // Explicitly set the constructor back-reference

    /* method definitions go here */
};
```

Another common technique that you are likely to see in older JavaScript code is to use the predefined prototype object with its `constructor` property and add methods to it one at a time with code like this:

```
// 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.toString = function() {
    return "(" + this.from + "..." + this.to + ")";
};
```

9.3 Classes with the class Keyword

Classes have been part of JavaScript since the very first version of the language, but in ES6, they finally got their own syntax with the introduction of the `class` keyword. Example 9-3 shows what our Range class looks like when written with this new syntax.

Example 9-3. The Range class rewritten using class

```
class Range {
    constructor(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;
    }

    // Return true if x is in the range, false otherwise
    // This method works for textual and Date ranges as well as numeric.
    includes(x) { return this.from <= x && x <= this.to; }

    // A generator function that makes instances of the class iterable.
    // Note that it only works for numeric ranges.
    *[Symbol.iterator]() {
        for(let x = Math.ceil(this.from); x <= this.to; x++) yield x;
    }

    // Return a string representation of the range
    toString() { return `(${this.from}...${this.to})`; }
}
```

```
// Here are example uses of this new Range class
let r = new Range(1,3);   // Create a Range object
r.includes(2)             // => true: 2 is in the range
r.toString()              // => "(1...3)"
[...r]                    // => [1, 2, 3]; convert to an array via iterator
```

It is important to understand that the classes defined in Examples 9-2 and 9-3 work in exactly the same way. The introduction of the `class` keyword to the language does not alter the fundamental nature of JavaScript's prototype-based classes. And although Example 9-3 uses the `class` keyword, the resulting Range object is a constructor function, just like the version defined in Example 9-2. The new `class` syntax is clean and convenient but is best thought of as "syntactic sugar" for the more fundamental class definition mechanism shown in Example 9-2.

Note the following things about the class syntax in Example 9-3:

- The class is declared with the `class` keyword, which is followed by the name of class and a class body in curly braces.
- The class body includes method definitions that use object literal method shorthand (which we also used in Example 9-1), where the `function` keyword is omitted. Unlike object literals, however, no commas are used to separate the methods from each other. (Although class bodies are superficially similar to object literals, they are not the same thing. In particular, they do not support the definition of properties with name/value pairs.)
- The keyword `constructor` is used to define the constructor function for the class. The function defined is not actually named "constructor", however. The `class` declaration statement defines a new variable `Range` and assigns the value of this special `constructor` function to that variable.
- If your class does not need to do any initialization, you can omit the `constructor` keyword and its body, and an empty constructor function will be implicitly created for you.

If you want to define a class that subclasses—or *inherits from*—another class, you can use the `extends` keyword with the `class` keyword:

```
// A Span is like a Range, but instead of initializing it with
// a start and an end, we initialize it with a start and a length
class Span extends Range {
    constructor(start, length) {
        if (length >= 0) {
            super(start, start + length);
        } else {
            super(start + length, start);
        }
    }
}
```

Creating subclasses is a whole topic of its own. We'll return to it, and explain the `extends` and `super` keywords shown here, in §9.5.

Like function declarations, class declarations have both statement and expression forms. Just as we can write:

```
let square = function(x) { return x * x; };
square(3)  // => 9
```

we can also write:

```
let Square = class { constructor(x) { this.area = x * x; } };
new Square(3).area  // => 9
```

As with function definition expressions, class definition expressions can include an optional class name. If you provide such a name, that name is only defined within the class body itself.

Although function expressions are quite common (particularly with the arrow function shorthand), in JavaScript programming, class definition expressions are not something that you are likely to use much unless you find yourself writing a function that takes a class as its argument and returns a subclass.

We'll conclude this introduction to the `class` keyword by mentioning a couple of important things you should know that are not apparent from `class` syntax:

- All code within the body of a `class` declaration is implicitly in strict mode (§5.6.3), even if no `"use strict"` directive appears. This means, for example, that you can't use octal integer literals or the `with` statement within class bodies and that you are more likely to get syntax errors if you forget to declare a variable before using it.
- Unlike function declarations, class declarations are not "hoisted." Recall from §8.1.1 that function definitions behave as if they had been moved to the top of the enclosing file or enclosing function, meaning that you can invoke a function in code that comes before the actual definition of the function. Although class declarations are like function declarations in some ways, they do not share this hoisting behavior: you *cannot* instantiate a class before you declare it.

9.3.1 Static Methods

You can define a static method within a `class` body by prefixing the method declaration with the `static` keyword. Static methods are defined as properties of the constructor function rather than properties of the prototype object.

For example, suppose we added the following code to Example 9-3:

```
    static parse(s) {
        let matches = s.match(/^\((\d+)\.\.\.(\d+)\)$/);
        if (!matches) {
            throw new TypeError(`Cannot parse Range from "${s}".`)
        }
        return new Range(parseInt(matches[1]), parseInt(matches[2]));
    }
```

The method defined by this code is `Range.parse()`, not `Range.prototype.parse()`, and you must invoke it through the constructor, not through an instance:

```
let r = Range.parse('(1...10)'); // Returns a new Range object
r.parse('(1...10)');             // TypeError: r.parse is not a function
```

You'll sometimes see static methods called *class methods* because they are invoked using the name of the class/constructor. When this term is used, it is to contrast class methods with the regular *instance methods* that are invoked on instances of the class. Because static methods are invoked on the constructor rather than on any particular instance, it almost never makes sense to use the `this` keyword in a static method.

We'll see examples of static methods in Example 9-4.

9.3.2 Getters, Setters, and other Method Forms

Within a `class` body, you can define getter and setter methods (§6.10.6) just as you can in object literals. The only difference is that in class bodies, you don't put a comma after the getter or setter. Example 9-4 includes a practical example of a getter method in a class.

In general, all of the shorthand method definition syntaxes allowed in object literals are also allowed in class bodies. This includes generator methods (marked with `*`) and methods whose names are the value of an expression in square brackets. In fact, you've already seen (in Example 9-3) a generator method with a computed name that makes the Range class iterable:

```
*[Symbol.iterator]() {
    for(let x = Math.ceil(this.from); x <= this.to; x++) yield x;
}
```

9.3.3 Public, Private, and Static Fields

In the discussion here of classes defined with the `class` keyword, we have only described the definition of methods within the class body. The ES6 standard only allows the creation of methods (including getters, setters, and generators) and static methods; it does not include syntax for defining fields. If you want to define a field (which is just an object-oriented synonym for "property") on a class instance, you must do that in the constructor function or in one of the methods. And if you want to

define a static field for a class, you must do that outside the class body, after the class has been defined. Example 9-4 includes examples of both kinds of fields.

Standardization is underway, however, for extended class syntax that allows the definition of instance and static fields, in both public and private forms. The code shown in the rest of this section is not yet standard JavaScript as of early 2020 but is already supported in Chrome and partially supported (public instance fields only) in Firefox. The syntax for public instance fields is in common use by JavaScript programmers using the React framework and the Babel transpiler.

Suppose you're writing a class like this one, with a constructor that initializes three fields:

```
class Buffer {
    constructor() {
        this.size = 0;
        this.capacity = 4096;
        this.buffer = new Uint8Array(this.capacity);
    }
}
```

With the new instance field syntax that is likely to be standardized, you could instead write:

```
class Buffer {
    size = 0;
    capacity = 4096;
    buffer = new Uint8Array(this.capacity);
}
```

The field initialization code has moved out of the constructor and now appears directly in the class body. (That code is still run as part of the constructor, of course. If you do not define a constructor, the fields are initialized as part of the implicitly created constructor.) The `this.` prefixes that appeared on the lefthand side of the assignments are gone, but note that you still must use `this.` to refer to these fields, even on the righthand side of the initializer assignments. The advantage of initializing your instance fields in this way is that this syntax allows (but does not require) you to put the initializers up at the top of the class definition, making it clear to readers exactly what fields will hold the state of each instance. You can declare fields without an initializer by just writing the name of the field followed by a semicolon. If you do that, the initial value of the field will be `undefined`. It is better style to always make the initial value explicit for all of your class fields.

Before the addition of this field syntax, class bodies looked a lot like object literals using shortcut method syntax, except that the commas had been removed. This field syntax—with equals signs and semicolons instead of colons and commas—makes it clear that class bodies are not at all the same as object literals.

The same proposal that seeks to standardize these instance fields also defines private instance fields. If you use the instance field initialization syntax shown in the previous example to define a field whose name begins with # (which is not normally a legal character in JavaScript identifiers), that field will be usable (with the # prefix) within the class body but will be invisible and inaccessible (and therefore immutable) to any code outside of the class body. If, for the preceding hypothetical Buffer class, you wanted to ensure that users of the class could not inadvertently modify the `size` field of an instance, you could use a private `#size` field instead, then define a getter function to provide read-only access to the value:

```
class Buffer {
    #size = 0;
    get size() { return this.#size; }
}
```

Note that private fields must be declared using this new field syntax before they can be used. You can't just write `this.#size = 0;` in the constructor of a class unless you include a "declaration" of the field directly in the class body.

Finally, a related proposal seeks to standardize the use of the `static` keyword for fields. If you add `static` before a public or private field declaration, those fields will be created as properties of the constructor function instead of properties of instances. Consider the static `Range.parse()` method we've defined. It included a fairly complex regular expression that might be good to factor out into its own static field. With the proposed new static field syntax, we could do that like this:

```
static integerRangePattern = /^\((\d+)\.\.\.(\d+)\)$/;
static parse(s) {
    let matches = s.match(Range.integerRangePattern);
    if (!matches) {
        throw new TypeError(`Cannot parse Range from "${s}".`)
    }
    return new Range(parseInt(matches[1]), matches[2]);
}
```

If we wanted this static field to be accessible only within the class, we could make it private using a name like `#pattern`.

9.3.4 Example: A Complex Number Class

Example 9-4 defines a class to represent complex numbers. The class is a relatively simple one, but it includes instance methods (including getters), static methods, instance fields, and static fields. It includes some commented-out code demonstrating how we might use the not-yet-standard syntax for defining instance fields and static fields within the class body.

Example 9-4. Complex.js: a complex number class

```
/**
 * Instances of this Complex class 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.
 */
class Complex {
    // Once class field declarations are standardized, we could declare
    // private fields to hold the real and imaginary parts of a complex number
    // here, with code like this:
    //
    // #r = 0;
    // #i = 0;

    // 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.
    constructor(real, imaginary) {
        this.r = real;       // This field holds the real part of the number.
        this.i = imaginary;  // This field holds the imaginary part.
    }

    // Here are two instance methods for addition and multiplication
    // of complex numbers. If c and d are instances of this class, we
    // might write c.plus(d) or d.times(c)
    plus(that) {
        return new Complex(this.r + that.r, this.i + that.i);
    }
    times(that) {
        return new Complex(this.r * that.r - this.i * that.i,
                           this.r * that.i + this.i * that.r);
    }

    // And here are static variants of the complex arithmetic methods.
    // We could write Complex.sum(c,d) and Complex.product(c,d)
    static sum(c, d) { return c.plus(d); }
    static product(c, d) { return c.times(d); }

    // These are some instance methods that are defined as getters
    // so they're used like fields. The real and imaginary getters would
    // be useful if we were using private fields this.#r and this.#i
    get real() { return this.r; }
    get imaginary() { return this.i; }
    get magnitude() { return Math.hypot(this.r, this.i); }

    // Classes should almost always have a toString() method
    toString() { return `{${this.r},${this.i}}`; }

    // It is often useful to define a method for testing whether
    // two instances of your class represent the same value
    equals(that) {
```

```
        return that instanceof Complex &&
            this.r === that.r &&
            this.i === that.i;
    }

    // Once static fields are supported inside class bodies, we could
    // define a useful Complex.ZERO constant like this:
    // static ZERO = new Complex(0,0);
}

// Here are some class fields that hold useful predefined complex numbers.
Complex.ZERO = new Complex(0,0);
Complex.ONE = new Complex(1,0);
Complex.I = new Complex(0,1);
```

With the Complex class of Example 9-4 defined, we can use the constructor, instance fields, instance methods, class fields, and class methods with code like this:

```
let c = new Complex(2, 3);     // Create a new object with the constructor
let d = new Complex(c.i, c.r); // Use instance fields of c
c.plus(d).toString()           // => "{5,5}"; use instance methods
c.magnitude                    // => Math.hypot(2,3); use a getter function
Complex.product(c, d)          // => new Complex(0, 13); a static method
Complex.ZERO.toString()        // => "{0,0}"; a static property
```

9.4 Adding Methods to Existing 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, for example, is code that adds a method for computing the complex conjugate to the Complex class of Example 9-4:

```
// 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. This is useful for implementing new language features in older versions of the language:

```
// If the new String method startsWith() is not already defined...
if (!String.prototype.startsWith) {
    // ...then define it like this using the older indexOf() method.
    String.prototype.startsWith = function(s) {
        return this.indexOf(s) === 0;
    };
}
```

Here is another example:

```
// Invoke the function f this many times, passing the iteration number
// For example, to print "hello" 3 times:
//     let n = 3;
//     n.times(i => { console.log(`hello ${i}`); });
Number.prototype.times = function(f, context) {
    let n = this.valueOf();
    for(let i = 0; i < n; i++) f.call(context, i);
};
```

Adding methods to the prototypes of built-in types like this is generally considered to be a bad idea because it will cause confusion and compatibility problems in the future if a new version of JavaScript defines a method with the same name. It is even possible to add methods to `Object.prototype`, making them available for all objects. But this is never a good thing to do because properties added to `Object.prototype` are visible to `for/in` loops (though you can avoid this by using `Object.defineProperty()` [§14.1] to make the new property non-enumerable).

9.5 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 the methods of A. The class B can define its own 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 often needs to invoke the overridden method in A. Similarly, the subclass constructor `B()` must typically invoke the superclass constructor `A()` in order to ensure that instances are completely initialized.

This section starts by showing how to define subclasses the old, pre-ES6 way, and then quickly moves on to demonstrate subclassing using the `class` and `extends` keywords and superclass constructor method invocation with the `super` keyword. Next is a subsection about avoiding subclasses and relying on object composition instead of inheritance. The section ends with an extended example that defines a hierarchy of Set classes and demonstrates how abstract classes can be used to separate interface from implementation.

9.5.1 Subclasses and Prototypes

Suppose we wanted to define a Span subclass of the Range class from Example 9-2. This subclass will work just like a Range, but instead of initializing it with a start and an end, we'll instead specify a start and a distance, or span. An instance of this Span class is also an instance of the Range superclass. A span instance inherits a customized `toString()` method from `Span.prototype`, but in order to be a subclass of Range, it must also inherit methods (such as `includes()`) from `Range.prototype`.

Example 9-5. Span.js: a simple subclass of Range

```
// This is the constructor function for our subclass
function Span(start, span) {
    if (span >= 0) {
        this.from = start;
        this.to = start + span;
    } else {
        this.to = start;
        this.from = start + span;
    }
}

// Ensure that the Span prototype inherits from the Range prototype
Span.prototype = Object.create(Range.prototype);

// We don't want to inherit Range.prototype.constructor, so we
// define our own constructor property.
Span.prototype.constructor = Span;

// By defining its own toString() method, Span overrides the
// toString() method that it would otherwise inherit from Range.
Span.prototype.toString = function() {
    return `(${this.from}... +${this.to - this.from})`;
};
```

In order to make Span a subclass of Range, we need to arrange for `Span.prototype` to inherit from `Range.prototype`. The key line of code in the preceding example is this one, and if it makes sense to you, you understand how subclasses work in JavaScript:

```
Span.prototype = Object.create(Range.prototype);
```

Objects created with the `Span()` constructor will inherit from the `Span.prototype` object. But we created that object to inherit from `Range.prototype`, so Span objects will inherit from both `Span.prototype` and `Range.prototype`.

You may notice that our `Span()` constructor sets the same `from` and `to` properties that the `Range()` constructor does and so does not need to invoke the `Range()` constructor to initialize the new object. Similarly, Span's `toString()` method completely reimplements the string conversion without needing to call Range's version of `toString()`. This makes Span a special case, and we can only really get away with this kind of subclassing because we know the implementation details of the superclass. A robust subclassing mechanism needs to allow classes to invoke the methods and constructor of their superclass, but prior to ES6, JavaScript did not have a simple way to do these things.

Fortunately, ES6 solves these problems with the `super` keyword as part of the `class` syntax. The next section demonstrates how it works.

9.5.2 Subclasses with extends and super

In ES6 and later, you can create a superclass simply by adding an `extends` clause to a class declaration, and you can do this even for built-in classes:

```
// A trivial Array subclass that adds getters for the first and last elements.
class EZArray extends Array {
    get first() { return this[0]; }
    get last() { return this[this.length-1]; }
}

let a = new EZArray();
a instanceof EZArray  // => true: a is subclass instance
a instanceof Array    // => true: a is also a superclass instance.
a.push(1,2,3,4);      // a.length == 4; we can use inherited methods
a.pop()               // => 4: another inherited method
a.first               // => 1: first getter defined by subclass
a.last                // => 3: last getter defined by subclass
a[1]                  // => 2: regular array access syntax still works.
Array.isArray(a)      // => true: subclass instance really is an array
EZArray.isArray(a)    // => true: subclass inherits static methods, too!
```

This EZArray subclass defines two simple getter methods. Instances of EZArray behave like ordinary arrays, and we can use inherited methods and properties like `push()`, `pop()`, and `length`. But we can also use the `first` and `last` getters defined in the subclass. Not only are instance methods like `pop()` inherited, but static methods like `Array.isArray` are also inherited. This is a new feature enabled by ES6 class syntax: `EZArray()` is a function, but it inherits from `Array()`:

```
// EZArray inherits instance methods because EZArray.prototype
// inherits from Array.prototype
Array.prototype.isPrototypeOf(EZArray.prototype) // => true

// And EZArray inherits static methods and properties because
// EZArray inherits from Array. This is a special feature of the
// extends keyword and is not possible before ES6.
Array.isPrototypeOf(EZArray) // => true
```

Our EZArray subclass is too simple to be very instructive. Example 9-6 is a more fully fleshed-out example. It defines a TypedMap subclass of the built-in Map class that adds type checking to ensure that the keys and values of the map are of the specified types (according to `typeof`). Importantly, this example demonstrates the use of the `super` keyword to invoke the constructor and methods of the superclass.

Example 9-6. TypedMap.js: a subclass of Map that checks key and value types

```
class TypedMap extends Map {
    constructor(keyType, valueType, entries) {
        // If entries are specified, check their types
        if (entries) {
```

```
            for(let [k, v] of entries) {
                if (typeof k !== keyType || typeof v !== valueType) {
                    throw new TypeError(`Wrong type for entry [${k}, ${v}]`);
                }
            }
        }

        // Initialize the superclass with the (type-checked) initial entries
        super(entries);

        // And then initialize this subclass by storing the types
        this.keyType = keyType;
        this.valueType = valueType;
    }

    // Now redefine the set() method to add type checking for any
    // new entries added to the map.
    set(key, value) {
        // Throw an error if the key or value are of the wrong type
        if (this.keyType && typeof key !== this.keyType) {
            throw new TypeError(`${key} is not of type ${this.keyType}`);
        }
        if (this.valueType && typeof value !== this.valueType) {
            throw new TypeError(`${value} is not of type ${this.valueType}`);
        }

        // If the types are correct, we invoke the superclass's version of
        // the set() method, to actually add the entry to the map. And we
        // return whatever the superclass method returns.
        return super.set(key, value);
    }
}
```

The first two arguments to the `TypedMap()` constructor are the desired key and value types. These should be strings, such as "number" and "boolean", that the `typeof` operator returns. You can also specify a third argument: an array (or any iterable object) of `[key,value]` arrays that specify the initial entries in the map. If you specify any initial entries, then the first thing the constructor does is verify that their types are correct. Next, the constructor invokes the superclass constructor, using the `super` keyword as if it was a function name. The `Map()` constructor takes one optional argument: an iterable object of `[key,value]` arrays. So the optional third argument of the `TypedMap()` constructor is the optional first argument to the `Map()` constructor, and we pass it to that superclass constructor with `super(entries)`.

After invoking the superclass constructor to initialize superclass state, the `TypedMap()` constructor next initializes its own subclass state by setting `this.keyType` and `this.valueType` to the specified types. It needs to set these properties so that it can use them again in the `set()` method.

There are a few important rules that you will need to know about using `super()` in constructors:

- If you define a class with the `extends` keyword, then the constructor for your class must use `super()` to invoke the superclass constructor.
- If you don't define a constructor in your subclass, one will be defined automatically for you. This implicitly defined constructor simply takes whatever values are passed to it and passes those values to `super()`.
- You may not use the `this` keyword in your constructor until after you have invoked the superclass constructor with `super()`. This enforces a rule that superclasses get to initialize themselves before subclasses do.
- The special expression `new.target` is undefined in functions that are invoked without the `new` keyword. In constructor functions, however, `new.target` is a reference to the constructor that was invoked. When a subclass constructor is invoked and uses `super()` to invoke the superclass constructor, that superclass constructor will see the subclass constructor as the value of `new.target`. A well-designed superclass should not need to know whether it has been subclassed, but it might be useful to be able to use `new.target.name` in logging messages, for example.

After the constructor, the next part of Example 9-6 is a method named `set()`. The Map superclass defines a method named `set()` to add a new entry to the map. We say that this `set()` method in TypedMap *overrides* the `set()` method of its superclass. This simple TypedMap subclass doesn't know anything about adding new entries to map, but it does know how to check types, so that is what it does first, verifying that the key and value to be added to the map have the correct types and throwing an error if they do not. This `set()` method doesn't have any way to add the key and value to the map itself, but that is what the superclass `set()` method is for. So we use the `super` keyword again to invoke the superclass's version of the method. In this context, `super` works much like the `this` keyword does: it refers to the current object but allows access to overridden methods defined in the superclass.

In constructors, you are required to invoke the superclass constructor before you can access `this` and initialize the new object yourself. There are no such rules when you override a method. A method that overrides a superclass method is not required to invoke the superclass method. If it does use `super` to invoke the overridden method (or any method) in the superclass, it can do that at the beginning or the middle or the end of the overriding method.

Finally, before we leave the TypedMap example behind, it is worth noting that this class is an ideal candidate for the use of private fields. As the class is written now, a user could change the `keyType` or `valueType` properties to subvert the type checking.

Once private fields are supported, we could change these properties to `#keyType` and `#valueType` so that they could not be altered from the outside.

9.5.3 Delegation Instead of Inheritance

The `extends` keyword makes it easy to create subclasses. But that does not mean that you *should* create lots of subclasses. If you want to write a class that shares the behavior of some other class, you can try to inherit that behavior by creating a subclass. But it is often easier and more flexible to get that desired behavior into your class by having your class create an instance of the other class and simply delegating to that instance as needed. You create a new class not by subclassing, but instead by wrapping or "composing" other classes. This delegation approach is often called "composition," and it is an oft-quoted maxim of object-oriented programming that one should "favor composition over inheritance."[2]

Suppose, for example, we wanted a Histogram class that behaves something like JavaScript's Set class, except that instead of just keeping track of whether a value has been added to set or not, it instead maintains a count of the number of times the value has been added. Because the API for this Histogram class is similar to Set, we might consider subclassing Set and adding a `count()` method. On the other hand, once we start thinking about how we might implement this `count()` method, we might realize that the Histogram class is more like a Map than a Set because it needs to maintain a mapping between values and the number of times they have been added. So instead of subclassing Set, we can create a class that defines a Set-like API but implements those methods by delegating to an internal Map object. Example 9-7 shows how we could do this.

Example 9-7. Histogram.js: a Set-like class implemented with delegation

```
/**
 * A Set-like class that keeps track of how many times a value has
 * been added. Call add() and remove() like you would for a Set, and
 * call count() to find out how many times a given value has been added.
 * The default iterator yields the values that have been added at least
 * once. Use entries() if you want to iterate [value, count] pairs.
 */
class Histogram {
    // To initialize, we just create a Map object to delegate to
    constructor() { this.map = new Map(); }

    // For any given key, the count is the value in the Map, or zero
    // if the key does not appear in the Map.
```

2 See *Design Patterns* (Addison-Wesley Professional) by Erich Gamma et al. or *Effective Java* (Addison-Wesley Professional) by Joshua Bloch, for example.

```
    count(key) { return this.map.get(key) || 0; }

    // The Set-like method has() returns true if the count is non-zero
    has(key) { return this.count(key) > 0; }

    // The size of the histogram is just the number of entries in the Map.
    get size() { return this.map.size; }

    // To add a key, just increment its count in the Map.
    add(key) { this.map.set(key, this.count(key) + 1); }

    // Deleting a key is a little trickier because we have to delete
    // the key from the Map if the count goes back down to zero.
    delete(key) {
        let count = this.count(key);
        if (count === 1) {
            this.map.delete(key);
        } else if (count > 1) {
            this.map.set(key, count - 1);
        }
    }

    // Iterating a Histogram just returns the keys stored in it
    [Symbol.iterator]() { return this.map.keys(); }

    // These other iterator methods just delegate to the Map object
    keys() { return this.map.keys(); }
    values() { return this.map.values(); }
    entries() { return this.map.entries(); }
}
```

All the `Histogram()` constructor does in Example 9-7 is create a Map object. And most of the methods are one-liners that just delegate to a method of the map, making the implementation quite simple. Because we used delegation rather than inheritance, a Histogram object is not an instance of Set or Map. But Histogram implements a number of commonly used Set methods, and in an untyped language like JavaScript, that is often good enough: a formal inheritance relationship is sometimes nice, but often optional.

9.5.4 Class Hierarchies and Abstract Classes

Example 9-6 demonstrated how we can subclass Map. Example 9-7 demonstrated how we can instead delegate to a Map object without actually subclassing anything. Using JavaScript classes to encapsulate data and modularize your code is often a great technique, and you may find yourself using the `class` keyword frequently. But you may find that you prefer composition to inheritance and that you rarely need to use `extends` (except when you're using a library or framework that requires you to extend its base classes).

There are some circumstances when multiple levels of subclassing are appropriate, however, and we'll end this chapter with an extended example that demonstrates a hierarchy of classes representing different kinds of sets. (The set classes defined in Example 9-8 are similar to, but not completely compatible with, JavaScript's built-in Set class.)

Example 9-8 defines lots of subclasses, but it also demonstrates how you can define *abstract classes*—classes that do not include a complete implementation—to serve as a common superclass for a group of related subclasses. An abstract superclass can define a partial implementation that all subclasses inherit and share. The subclasses, then, only need to define their own unique behavior by implementing the abstract methods defined—but not implemented—by the superclass. Note that JavaScript does not have any formal definition of abstract methods or abstract classes; I'm simply using that name here for unimplemented methods and incompletely implemented classes.

Example 9-8 is well commented and stands on its own. I encourage you to read it as a capstone example for this chapter on classes. The final class in Example 9-8 does a lot of bit manipulation with the `&`, `|`, and `~` operators, which you can review in §4.8.3.

Example 9-8. Sets.js: a hierarchy of abstract and concrete set classes

```
/**
 * The AbstractSet class defines a single abstract method, has().
 */
class AbstractSet {
    // Throw an error here so that subclasses are forced
    // to define their own working version of this method.
    has(x) { throw new Error("Abstract method"); }
}

/**
 * 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 and convert it to a
 * string using mathematical notation.
 */
class NotSet extends AbstractSet {
    constructor(set) {
        super();
        this.set = set;
    }

    // Our implementation of the abstract method we inherited
    has(x) { return !this.set.has(x); }
    // And we also override this Object method
```

```
    toString() { return `{ x| x ∉ ${this.set.toString()} }`; }
}

/**
 * Range set is a concrete subclass of AbstractSet. Its members are
 * all values that are between the from and to bounds, inclusive.
 * Since its members can be floating point numbers, it is not
 * enumerable and does not have a meaningful size.
 */
class RangeSet extends AbstractSet {
    constructor(from, to) {
        super();
        this.from = from;
        this.to = to;
    }

    has(x) { return x >= this.from && x <= this.to; }
    toString() { return `{ x| ${this.from} ≤ x ≤ ${this.to} }`; }
}

/*
 * AbstractEnumerableSet is an abstract subclass of AbstractSet.  It defines
 * an abstract getter that returns the size of the set and also defines an
 * abstract iterator. And it then implements concrete isEmpty(), toString(),
 * and equals() methods on top of those. Subclasses that implement the
 * iterator, the size getter, and the has() method get these concrete
 * methods for free.
 */
class AbstractEnumerableSet extends AbstractSet {
    get size() { throw new Error("Abstract method"); }
    [Symbol.iterator]() { throw new Error("Abstract method"); }

    isEmpty() { return this.size === 0; }
    toString() { return `{${Array.from(this).join(", ")}}`; }
    equals(set) {
        // If the other set is not also Enumerable, it isn't equal to this one
        if (!(set instanceof AbstractEnumerableSet)) return false;

        // If they don't have the same size, they're not equal
        if (this.size !== set.size) return false;

        // Loop through the elements of this set
        for(let element of this) {
            // If an element isn't in the other set, they aren't equal
            if (!set.has(element)) return false;
        }

        // The elements matched, so the sets are equal
        return true;
    }
}
```

```
/*
 * SingletonSet is a concrete subclass of AbstractEnumerableSet.
 * A singleton set is a read-only set with a single member.
 */
class SingletonSet extends AbstractEnumerableSet {
    constructor(member) {
        super();
        this.member = member;
    }

    // We implement these three methods, and inherit isEmpty, equals()
    // and toString() implementations based on these methods.
    has(x) { return x === this.member; }
    get size() { return 1; }
    *[Symbol.iterator]() { yield this.member; }
}

/*
 * AbstractWritableSet is an abstract subclass of AbstractEnumerableSet.
 * It defines the abstract methods insert() and remove() that insert and
 * remove individual elements from the set, and then implements concrete
 * add(), subtract(), and intersect() methods on top of those. Note that
 * our API diverges here from the standard JavaScript Set class.
 */
class AbstractWritableSet extends  AbstractEnumerableSet {
    insert(x) { throw new Error("Abstract method"); }
    remove(x) { throw new Error("Abstract method"); }

    add(set) {
        for(let element of set) {
            this.insert(element);
        }
    }

    subtract(set) {
        for(let element of set) {
            this.remove(element);
        }
    }

    intersect(set) {
        for(let element of this) {
            if (!set.has(element)) {
                this.remove(element);
            }
        }
    }
}

/**
 * A BitSet is a concrete subclass of AbstractWritableSet with a
 * very efficient fixed-size set implementation for sets whose
```

```
 * elements are non-negative integers less than some maximum size.
 */
class BitSet extends AbstractWritableSet {
    constructor(max) {
        super();
        this.max = max;  // The maximum integer we can store.
        this.n = 0;      // How many integers are in the set
        this.numBytes = Math.floor(max / 8) + 1;   // How many bytes we need
        this.data = new Uint8Array(this.numBytes); // The bytes
    }

    // Internal method to check if a value is a legal member of this set
    _valid(x) { return Number.isInteger(x) && x >= 0 && x <= this.max; }

    // Tests whether the specified bit of the specified byte of our
    // data array is set or not. Returns true or false.
    _has(byte, bit) { return (this.data[byte] & BitSet.bits[bit]) !== 0; }

    // Is the value x in this BitSet?
    has(x) {
        if (this._valid(x)) {
            let byte = Math.floor(x / 8);
            let bit = x % 8;
            return this._has(byte, bit);
        } else {
            return false;
        }
    }

    // Insert the value x into the BitSet
    insert(x) {
        if (this._valid(x)) {               // If the value is valid
            let byte = Math.floor(x / 8);   // convert to byte and bit
            let bit = x % 8;
            if (!this._has(byte, bit)) {    // If that bit is not set yet
                this.data[byte] |= BitSet.bits[bit]; // then set it
                this.n++;                            // and increment set size
            }
        } else {
            throw new TypeError("Invalid set element: " + x );
        }
    }

    remove(x) {
        if (this._valid(x)) {              // If the value is valid
            let byte = Math.floor(x / 8);  // compute the byte and bit
            let bit = x % 8;
            if (this._has(byte, bit)) {    // If that bit is already set
                this.data[byte] &= BitSet.masks[bit];  // then unset it
                this.n--;                              // and decrement size
            }
        } else {
```

```
            throw new TypeError("Invalid set element: " + x );
        }
    }

    // A getter to return the size of the set
    get size() { return this.n; }

    // Iterate the set by just checking each bit in turn.
    // (We could be a lot more clever and optimize this substantially)
    *[Symbol.iterator]() {
        for(let i = 0; i <= this.max; i++) {
            if (this.has(i)) {
                yield i;
            }
        }
    }
}

// Some pre-computed values used by the has(), insert() and remove() methods
BitSet.bits = new Uint8Array([1, 2, 4, 8, 16, 32, 64, 128]);
BitSet.masks = new Uint8Array([~1, ~2, ~4, ~8, ~16, ~32, ~64, ~128]);
```

9.6 Summary

This chapter has explained the key features of JavaScript classes:

- Objects that are members of the same class inherit properties from the same prototype object. The prototype object is the key feature of JavaScript classes, and it is possible to define classes with nothing more than the `Object.create()` method.
- Prior to ES6, classes were more typically defined by first defining a constructor function. Functions created with the `function` keyword have a `prototype` property, and the value of this property is an object that is used as the prototype of all objects created when the function is invoked with `new` as a constructor. By initializing this prototype object, you can define the shared methods of your class. Although the prototype object is the key feature of the class, the constructor function is the public identity of the class.
- ES6 introduces a `class` keyword that makes it easier to define classes, but under the hood, constructor and prototype mechanism remains the same.
- Subclasses are defined using the `extends` keyword in a class declaration.
- Subclasses can invoke the constructor of their superclass or overridden methods of their superclass with the `super` keyword.

CHAPTER 10
Modules

The goal of modular programming is to allow large programs to be assembled using modules of code from disparate authors and sources and for all of that code to run correctly even in the presence of code that the various module authors did not anticipate. As a practical matter, modularity is mostly about encapsulating or hiding private implementation details and keeping the global namespace tidy so that modules cannot accidentally modify the variables, functions, and classes defined by other modules.

Until recently, JavaScript had no built-in support for modules, and programmers working on large code bases did their best to use the weak modularity available through classes, objects, and closures. Closure-based modularity, with support from code-bundling tools, led to a practical form of modularity based on a `require()` function, which was adopted by Node. `require()`-based modules are a fundamental part of the Node programming environment but were never adopted as an official part of the JavaScript language. Instead, ES6 defines modules using `import` and `export` keywords. Although `import` and `export` have been part of the language for years, they were only implemented by web browsers and Node relatively recently. And, as a practical matter, JavaScript modularity still depends on code-bundling tools.

The sections that follow cover:

- Do-it-yourself modules with classes, objects, and closures
- Node modules using `require()`
- ES6 modules using `export`, `import`, and `import()`

10.1 Modules with Classes, Objects, and Closures

Though it may be obvious, it is worth pointing out that one of the important features of classes is that they act as modules for their methods. Think back to Example 9-8. That example defined a number of different classes, all of which had a method named `has()`. But you would have no problem writing a program that used multiple set classes from that example: there is no danger that the implementation of `has()` from SingletonSet will overwrite the `has()` method of BitSet, for example.

The reason that the methods of one class are independent of the methods of other, unrelated classes is that the methods of each class are defined as properties of independent prototype objects. The reason that classes are modular is that objects are modular: defining a property in a JavaScript object is a lot like declaring a variable, but adding properties to objects does not affect the global namespace of a program, nor does it affect the properties of other objects. JavaScript defines quite a few mathematical functions and constants, but instead of defining them all globally, they are grouped as properties of a single global Math object. This same technique could have been used in Example 9-8. Instead of defining global classes with names like SingletonSet and BitSet, that example could have been written to define only a single global Sets object, with properties referencing the various classes. Users of this Sets library could then refer to the classes with names like `Sets.Singleton` and `Sets.Bit`.

Using classes and objects for modularity is a common and useful technique in JavaScript programming, but it doesn't go far enough. In particular, it doesn't offer us any way to hide internal implementation details inside the module. Consider Example 9-8 again. If we were writing that example as a module, maybe we would have wanted to keep the various abstract classes internal to the module, only making the concrete subclasses available to users of the module. Similarly, in the BitSet class, the `_valid()` and `_has()` methods are internal utilities that should not really be exposed to users of the class. And `BitSet.bits` and `BitSet.masks` are implementation details that would be better off hidden.

As we saw in §8.6, local variables and nested functions declared within a function are private to that function. This means that we can use immediately invoked function expressions to achieve a kind of modularity by leaving the implementation details and utility functions hidden within the enclosing function but making the public API of the module the return value of the function. In the case of the BitSet class, we might structure the module like this:

```
const BitSet = (function() { // Set BitSet to the return value of this function
    // Private implementation details here
    function isValid(set, n) { ... }
    function has(set, byte, bit) { ... }
    const BITS = new Uint8Array([1, 2, 4, 8, 16, 32, 64, 128]);
    const MASKS = new Uint8Array([~1, ~2, ~4, ~8, ~16, ~32, ~64, ~128]);
```

```
    // The public API of the module is just the BitSet class, which we define
    // and return here. The class can use the private functions and constants
    // defined above, but they will be hidden from users of the class
    return class BitSet extends AbstractWritableSet {
        // ... implementation omitted ...
    };
}());
```

This approach to modularity becomes a little more interesting when the module has more than one item in it. The following code, for example, defines a mini statistics module that exports `mean()` and `stddev()` functions while leaving the implementation details hidden:

```
// This is how we could define a stats module
const stats = (function() {
    // Utility functions private to the module
    const sum = (x, y) => x + y;
    const square = x => x * x;

    // A public function that will be exported
    function mean(data) {
        return data.reduce(sum)/data.length;
    }

    // A public function that we will export
    function stddev(data) {
        let m = mean(data);
        return Math.sqrt(
            data.map(x => x - m).map(square).reduce(sum)/(data.length-1)
        );
    }

    // We export the public function as properties of an object
    return { mean, stddev };
}());

// And here is how we might use the module
stats.mean([1, 3, 5, 7, 9])   // => 5
stats.stddev([1, 3, 5, 7, 9]) // => Math.sqrt(10)
```

10.1.1 Automating Closure-Based Modularity

Note that it is a fairly mechanical process to transform a file of JavaScript code into this kind of module by inserting some text at the beginning and end of the file. All that is needed is some convention for the file of JavaScript code to indicate which values are to be exported and which are not.

Imagine a tool that takes a set of files, wraps the content of each of those files within an immediately invoked function expression, keeps track of the return value of each

function, and concatenates everything into one big file. The result might look something like this:

```
const modules = {};
function require(moduleName) { return modules[moduleName]; }

modules["sets.js"] = (function() {
    const exports = {};

    // The contents of the sets.js file go here:
    exports.BitSet = class BitSet { ... };

    return exports;
}());

modules["stats.js"] = (function() {
    const exports = {};

    // The contents of the stats.js file go here:
    const sum = (x, y) => x + y;
    const square = x = > x * x;
    exports.mean = function(data) { ... };
    exports.stddev = function(data) { ... };

    return exports;
}());
```

With modules bundled up into a single file like the one shown in the preceding example, you can imagine writing code like the following to make use of those modules:

```
// Get references to the modules (or the module content) that we need
const stats = require("stats.js");
const BitSet = require("sets.js").BitSet;

// Now write code using those modules
let s = new BitSet(100);
s.insert(10);
s.insert(20);
s.insert(30);
let average = stats.mean([...s]); // average is 20
```

This code is a rough sketch of how code-bundling tools (such as webpack and Parcel) for web browsers work, and it's also a simple introduction to the `require()` function like the one used in Node programs.

10.2 Modules in Node

In Node programming, it is normal to split programs into as many files as seems natural. These files of JavaScript code are assumed to all live on a fast filesystem. Unlike web browsers, which have to read files of JavaScript over a relatively slow network connection, there is no need or benefit to bundling a Node program into a single JavaScript file.

In Node, each file is an independent module with a private namespace. Constants, variables, functions, and classes defined in one file are private to that file unless the file exports them. And values exported by one module are only visible in another module if that module explicitly imports them.

Node modules import other modules with the `require()` function and export their public API by setting properties of the Exports object or by replacing the `module.exports`object entirely.

10.2.1 Node Exports

Node defines a global `exports` object that is always defined. If you are writing a Node module that exports multiple values, you can simply assign them to the properties of this object:

```
const sum = (x, y) => x + y;
const square = x => x * x;

exports.mean = data => data.reduce(sum)/data.length;
exports.stddev = function(d) {
    let m = exports.mean(d);
    return Math.sqrt(d.map(x => x - m).map(square).reduce(sum)/(d.length-1));
};
```

Often, however, you want to define a module that exports only a single function or class rather than an object full of functions or classes. To do this, you simply assign the single value you want to export to `module.exports`:

```
module.exports = class BitSet extends AbstractWritableSet {
    // implementation omitted
};
```

The default value of `module.exports` is the same object that `exports` refers to. In the previous stats module, we could have assigned the mean function to `module.exports.mean` instead of `exports.mean`. Another approach with modules like the stats module is to export a single object at the end of the module rather than exporting functions one by one as you go:

```
// Define all the functions, public and private
const sum = (x, y) => x + y;
const square = x => x * x;
```

```
const mean = data => data.reduce(sum)/data.length;
const stddev = d => {
    let m = mean(d);
    return Math.sqrt(d.map(x => x - m).map(square).reduce(sum)/(d.length-1));
};

// Now export only the public ones
module.exports = { mean, stddev };
```

10.2.2 Node Imports

A Node module imports another module by calling the `require()` function. The argument to this function is the name of the module to be imported, and the return value is whatever value (typically a function, class, or object) that module exports.

If you want to import a system module built in to Node or a module that you have installed on your system via a package manager, then you simply use the unqualified name of the module, without any "/" characters that would turn it into a filesystem path:

```
// These modules are built in to Node
const fs = require("fs");           // The built-in filesystem module
const http = require("http");       // The built-in HTTP module

// The Express HTTP server framework is a third-party module.
// It is not part of Node but has been installed locally
const express = require("express");
```

When you want to import a module of your own code, the module name should be the path to the file that contains that code, relative to the current module's file. It is legal to use absolute paths that begin with a / character, but typically, when importing modules that are part of your own program, the module names will begin with ./ or sometimes ../ to indicate that they are relative to the current directory or the parent directory. For example:

```
const stats = require('./stats.js');
const BitSet = require('./utils/bitset.js');
```

(You can also omit the *.js* suffix on the files you're importing and Node will still find the files, but it is common to see these file extensions explicitly included.)

When a module exports just a single function or class, all you have to do is require it. When a module exports an object with multiple properties, you have a choice: you can import the entire object, or just import the specific properties (using destructuring assignment) of the object that you plan to use. Compare these two approaches:

```
// Import the entire stats object, with all of its functions
const stats = require('./stats.js');

// We've got more functions than we need, but they're neatly
```

```
// organized into a convenient "stats" namespace.
let average = stats.mean(data);

// Alternatively, we can use idiomatic destructuring assignment to import
// exactly the functions we want directly into the local namespace:
const { stddev } = require('./stats.js');

// This is nice and succinct, though we lose a bit of context
// without the 'stats' prefix as a namspace for the stddev() function.
let sd = stddev(data);
```

10.2.3 Node-Style Modules on the Web

Modules with an Exports object and a `require()` function are built in to Node. But if you're willing to process your code with a bundling tool like webpack, then it is also possible to use this style of modules for code that is intended to run in web browsers. Until recently, this was a very common thing to do, and you may see lots of web-based code that still does it.

Now that JavaScript has its own standard module syntax, however, developers who use bundlers are more likely to use the official JavaScript modules with `import` and `export` statements.

10.3 Modules in ES6

ES6 adds `import` and `export` keywords to JavaScript and finally supports real modularity as a core language feature. ES6 modularity is conceptually the same as Node modularity: each file is its own module, and constants, variables, functions, and classes defined within a file are private to that module unless they are explicitly exported. Values that are exported from one module are available for use in modules that explicitly import them. ES6 modules differ from Node modules in the syntax used for exporting and importing and also in the way that modules are defined in web browsers. The sections that follow explain these things in detail.

First, though, note that ES6 modules are also different from regular JavaScript "scripts" in some important ways. The most obvious difference is the modularity itself: in regular scripts, top-level declarations of variables, functions, and classes go into a single global context shared by all scripts. With modules, each file has its own private context and can use the `import` and `export` statements, which is the whole point, after all. But there are other differences between modules and scripts as well. Code inside an ES6 module (like code inside any ES6 `class` definition) is automatically in strict mode (see §5.6.3). This means that, when you start using ES6 modules, you'll never have to write `"use strict"` again. And it means that code in modules cannot use the `with` statement or the `arguments` object or undeclared variables. ES6 modules are even slightly stricter than strict mode: in strict mode, in functions

invoked as functions, `this` is `undefined`. In modules, `this` is `undefined` even in top-level code. (By contrast, scripts in web browsers and Node set `this` to the global object.)

ES6 Modules on the Web and in Node

ES6 modules have been in use on the web for years with the help of code bundlers like webpack, which combine independent modules of JavaScript code into large, non-modular bundles suitable for inclusion into web pages. At the time of this writing, however, ES6 modules are finally supported natively by all web browsers other than Internet Explorer. When used natively, ES6 modules are added into HTML pages with a special `<script type="module">` tag, described later in this chapter.

And meanwhile, having pioneered JavaScript modularity, Node finds itself in the awkward position of having to support two not entirely compatible module systems. Node 13 supports ES6 modules, but for now, the vast majority of Node programs still use Node modules.

10.3.1 ES6 Exports

To export a constant, variable, function, or class from an ES6 module, simply add the keyword `export` before the declaration:

```
export const PI = Math.PI;

export function degreesToRadians(d) { return d * PI / 180; }

export class Circle {
    constructor(r) { this.r = r; }
    area() { return PI * this.r * this.r; }
}
```

As an alternative to scattering `export` keywords throughout your module, you can define your constants, variables, functions, and classes as you normally would, with no `export` statement, and then (typically at the end of your module) write a single `export` statement that declares exactly what is exported in a single place. So instead of writing three individual exports in the preceding code, we could have equivalently written a single line at the end:

```
export { Circle, degreesToRadians, PI };
```

This syntax looks like the `export` keyword followed by an object literal (using shorthand notation). But in this case, the curly braces do not actually define an object literal. This export syntax simply requires a comma-separated list of identifiers within curly braces.

It is common to write modules that export only one value (typically a function or class), and in this case, we usually use `export default` instead of `export`:

```
export default class BitSet {
    // implementation omitted
}
```

Default exports are slightly easier to import than non-default exports, so when there is only one exported value, using `export default` makes things easier for the modules that use your exported value.

Regular exports with `export` can only be done on declarations that have a name. Default exports with `export default` can export any expression including anonymous function expressions and anonymous class expressions. This means that if you use `export default`, you can export object literals. So unlike the `export` syntax, if you see curly braces after `export default`, it really is an object literal that is being exported.

It is legal, but somewhat uncommon, for modules to have a set of regular exports and also a default export. If a module has a default export, it can only have one.

Finally, note that the `export` keyword can only appear at the top level of your JavaScript code. You may not export a value from within a class, function, loop, or conditional. (This is an important feature of the ES6 module system and enables static analysis: a modules export will be the same on every run, and the symbols exported can be determined before the module is actually run.)

10.3.2 ES6 Imports

You import values that have been exported by other modules with the `import` keyword. The simplest form of import is used for modules that define a default export:

```
import BitSet from './bitset.js';
```

This is the `import` keyword, followed by an identifier, followed by the `from` keyword, followed by a string literal that names the module whose default export we are importing. The default export value of the specified module becomes the value of the specified identifier in the current module.

The identifier to which the imported value is assigned is a constant, as if it had been declared with the `const` keyword. Like exports, imports can only appear at the top level of a module and are not allowed within classes, functions, loops, or conditionals. By near-universal convention, the imports needed by a module are placed at the start of the module. Interestingly, however, this is not required: like function declarations, imports are "hoisted" to the top, and all imported values are available for any of the module's code runs.

The module from which a value is imported is specified as a constant string literal in single quotes or double quotes. (You may not use a variable or other expression whose value is a string, and you may not use a string within backticks because template literals can interpolate variables and do not always have constant values.) In web browsers, this string is interpreted as a URL relative to the location of the module that is doing the importing. (In Node, or when using a bundling tool, the string is interpreted as a filename relative to the current module, but this makes little difference in practice.) A *module specifier* string must be an absolute path starting with "/", or a relative path starting with "./" or "../", or a complete URL a with protocol and hostname. The ES6 specification does not allow unqualified module specifier strings like "util.js" because it is ambiguous whether this is intended to name a module in the same directory as the current one or some kind of system module that is installed in some special location. (This restriction against "bare module specifiers" is not honored by code-bundling tools like webpack, which can easily be configured to find bare modules in a library directory that you specify.) A future version of the language may allow "bare module specifiers," but for now, they are not allowed. If you want to import a module from the same directory as the current one, simply place "./" before the module name and import from "./util.js" instead of "util.js".

So far, we've only considered the case of importing a single value from a module that uses `export default`. To import values from a module that exports multiple values, we use a slightly different syntax:

```
import { mean, stddev } from "./stats.js";
```

Recall that default exports do not need to have a name in the module that defines them. Instead, we provide a local name when we import those values. But non-default exports of a module do have names in the exporting module, and when we import those values, we refer to them by those names. The exporting module can export any number of named value. An `import` statement that references that module can import any subset of those values simply by listing their names within curly braces. The curly braces make this kind of `import` statement look something like a destructuring assignment, and destructuring assignment is actually a good analogy for what this style of import is doing. The identifiers within curly braces are all hoisted to the top of the importing module and behave like constants.

Style guides sometimes recommend that you explicitly import every symbol that your module will use. When importing from a module that defines many exports, however, you can easily import everything with an `import` statement like this:

```
import * as stats from "./stats.js";
```

An `import` statement like this creates an object and assigns it to a constant named `stats`. Each of the non-default exports of the module being imported becomes a property of this `stats` object. Non-default exports always have names, and those are

used as property names within the object. Those properties are effectively constants: they cannot be overwritten or deleted. With the wildcard import shown in the previous example, the importing module would use the imported `mean()` and `stddev()` functions through the `stats` object, invoking them as `stats.mean()` and `stats.stddev()`.

Modules typically define either one default export or multiple named exports. It is legal, but somewhat uncommon, for a module to use both `export` and `export default`. But when a module does that, you can import both the default value and the named values with an `import` statement like this:

```
import Histogram, { mean, stddev } from "./histogram-stats.js";
```

So far, we've seen how to import from modules with a default export and from modules with non-default or named exports. But there is one other form of the `import` statement that is used with modules that have no exports at all. To include a no-exports module into your program, simply use the `import` keyword with the module specifier:

```
import "./analytics.js";
```

A module like this runs the first time it is imported. (And subsequent imports do nothing.) A module that just defines functions is only useful if it exports at least one of those functions. But if a module runs some code, then it can be useful to import even without symbols. An analytics module for a web application might run code to register various event handlers and then use those event handlers to send telemetry data back to the server at appropriate times. The module is self-contained and does not need to export anything, but we still need to `import` it so that it does actually run as part of our program.

Note that you can use this import-nothing `import` syntax even with modules that do have exports. If a module defines useful behavior independent of the values it exports, and if your program does not need any of those exported values, you can still import the module . just for that default behavior.

10.3.3 Imports and Exports with Renaming

If two modules export two different values using the same name and you want to import both of those values, you will have to rename one or both of the values when you import it. Similarly, if you want to import a value whose name is already in use in your module, you will need to rename the imported value. You can use the `as` keyword with named imports to rename them as you import them:

```
import { render as renderImage } from "./imageutils.js";
import { render as renderUI } from "./ui.js";
```

These lines import two functions into the current module. The functions are both named `render()` in the modules that define them but are imported with the more descriptive and disambiguating names `renderImage()` and `renderUI()`.

Recall that default exports do not have a name. The importing module always chooses the name when importing a default export. So there is no need for a special syntax for renaming in that case.

Having said that, however, the possibility of renaming on import provides another way of importing from modules that define both a default export and named exports. Recall the "./histogram-stats.js" module from the previous section. Here is another way to import both the default and named exports of that module:

```
import { default as Histogram, mean, stddev } from "./histogram-stats.js";
```

In this case, the JavaScript keyword `default` serves as a placeholder and allows us to indicate that we want to import and provide a name for the default export of the module.

It is also possible to rename values as you export them, but only when using the curly brace variant of the `export` statement. It is not common to need to do this, but if you chose short, succinct names for use inside your module, you might prefer to export your values with more descriptive names that are less likely to conflict with other modules. As with imports, you use the `as` keyword to do this:

```
export {
    layout as calculateLayout,
    render as renderLayout
};
```

Keep in mind that, although the curly braces look something like object literals, they are not, and the `export` keyword expects a single identifier before the `as`, not an expression. This means, unfortunately, that you cannot use export renaming like this:

```
export { Math.sin as sin, Math.cos as cos }; // SyntaxError
```

10.3.4 Re-Exports

Throughout this chapter, we've discussed a hypothetical "./stats.js" module that exports `mean()` and `stddev()` functions. If we were writing such a module and we thought that many users of the module would want only one function or the other, then we might want to define `mean()` in a "./stats/mean.js" module and define `stddev()` in "./stats/stddev.js". That way, programs only need to import exactly the functions they need and are not bloated by importing code they do not need.

Even if we had defined these statistical functions in individual modules, however, we might expect that there would be plenty of programs that want both functions and

would appreciate a convenient "./stats.js" module from which they could import both on one line.

Given that the implementations are now in separate files, defining this "./stat.js" module is simple:

```
import { mean } from "./stats/mean.js";
import { stddev } from "./stats/stddev.js";
export { mean, stdev };
```

ES6 modules anticipate this use case and provide a special syntax for it. Instead of importing a symbol simply to export it again, you can combine the import and the export steps into a single "re-export" statement that uses the `export` keyword and the `from` keyword:

```
export { mean } from "./stats/mean.js";
export { stddev } from "./stats/stddev.js";
```

Note that the names `mean` and `stddev` are not actually used in this code. If we are not being selective with a re-export and simply want to export all of the named values from another module, we can use a wildcard:

```
export * from "./stats/mean.js";
export * from "./stats/stddev.js";
```

Re-export syntax allows renaming with `as` just as regular `import` and `export` statements do. Suppose we wanted to re-export the `mean()` function but also define `average()` as another name for the function. We could do that like this:

```
export { mean, mean as average } from "./stats/mean.js";
export { stddev } from "./stats/stddev.js";
```

All of the re-exports in this example assume that the "./stats/mean.js" and "./stats/stddev.js" modules export their functions using `export` instead of `export default`. In fact, however, since these are modules with only a single export, it would have made sense to define them with `export default`. If we had done so, then the re-export syntax is a little more complicated because it needs to define a name for the unnamed default exports. We can do that like this:

```
export { default as mean } from "./stats/mean.js";
export { default as stddev } from "./stats/stddev.js";
```

If you want to re-export a named symbol from another module as the default export of your module, you could do an `import` followed by an `export default`, or you could combine the two statements like this:

```
// Import the mean() function from ./stats.js and make it the
// default export of this module
export { mean as default } from "./stats.js"
```

And finally, to re-export the default export of another module as the default export of your module (though it is unclear why you would want to do this, since users could simply import the other module directly), you can write:

```
// The average.js module simply re-exports the stats/mean.js default export
export { default } from "./stats/mean.js"
```

10.3.5 JavaScript Modules on the Web

The preceding sections have described ES6 modules and their `import` and `export` declarations in a somewhat abstract manner. In this section and the next, we'll be discussing how they actually work in web browsers, and if you are not already an experienced web developer, you may find the rest of this chapter easier to understand after you have read Chapter 15.

As of early 2020, production code using ES6 modules is still generally bundled with a tool like webpack. There are trade-offs to doing this,[1] but on the whole, code bundling tends to give better performance. That may well change in the future as network speeds grow and browser vendors continue to optimize their ES6 module implementations.

Even though bundling tools may still be desirable in production, they are no longer required in development since all current browsers provide native support for JavaScript modules. Recall that modules use strict mode by default, `this` does not refer to a global object, and top-level declarations are not shared globally by default. Since modules must be executed differently than legacy non-module code, their introduction requires changes to HTML as well as JavaScript. If you want to natively use `import` directives in a web browser, you must tell the web browser that your code is a module by using a `<script type="module">` tag.

One of the nice features of ES6 modules is that each module has a static set of imports. So given a single starting module, a web browser can load all of its imported modules and then load all of the modules imported by that first batch of modules, and so on, until a complete program has been loaded. We've seen that the module specifier in an `import` statement can be treated as a relative URL. A `<script type="module">` tag marks the starting point of a modular program. None of the modules it imports are expected to be in `<script>` tags, however: instead, they are loaded on demand as regular JavaScript files and are executed in strict mode as regular ES6 modules. Using a `<script type="module">` tag to define the main entry point for a modular JavaScript program can be as simple as this:

1 For example: web apps that have frequent incremental updates and users who make frequent return visits may find that using small modules instead of large bundles can result in better average load times because of better utilization of the user's browser cache.

```
<script type="module">import "./main.js";</script>
```

Code inside an inline `<script type="module">` tag is an ES6 module, and as such can use the `export` statement. There is not any point in doing so, however, because the HTML `<script>` tag syntax does not provide any way to define a name for inline modules, so even if such a module does export a value, there is no way for another module to import it.

Scripts with the `type="module"` attribute are loaded and executed like scripts with the `defer` attribute. Loading of the code begins as soon as the HTML parser encounters the `<script>` tag (in the case of modules, this code-loading step may be a recursive process that loads multiple JavaScript files). But code execution does not begin until HTML parsing is complete. And once HTML parsing is complete, scripts (both modular and non) are executed in the order in which they appear in the HTML document.

You can modify the execution time of modules with the `async` attribute, which works the same way for modules that it does for regular scripts. An `async` module will execute as soon as the code is loaded, even if HTML parsing is not complete and even if this changes the relative ordering of the scripts.

Web browsers that support `<script type="module">` must also support `<script nomodule>`. Browsers that are module-aware ignore any script with the `nomodule` attribute and will not execute it. Browsers that do not support modules will not recognize the `nomodule` attribute, so they will ignore it and run the script. This provides a powerful technique for dealing with browser compatibility issues. Browsers that support ES6 modules also support other modern JavaScript features like classes, arrow functions, and the `for/of` loop. If you write modern JavaScript and load it with `<script type="module">`, you know that it will only be loaded by browsers that can support it. And as a fallback for IE11 (which, in 2020, is effectively the only remaining browser that does not support ES6), you can use tools like Babel and webpack to transform your code into non-modular ES5 code, then load that less-efficient transformed code via `<script nomodule>`.

Another important difference between regular scripts and module scripts has to do with cross-origin loading. A regular `<script>` tag will load a file of JavaScript code from any server on the internet, and the internet's infrastructure of advertising, analytics, and tracking code depends on that fact. But `<script type="module">` provides an opportunity to tighten this up, and modules can only be loaded from the same origin as the containing HTML document or when proper CORS headers are in place to securely allow cross-origin loads. An unfortunate side effect of this new security restriction is that it makes it difficult to test ES6 modules in development mode using `file:` URLs. When using ES6 modules, you will likely need to set up a static web server for testing.

Some programmers like to use the filename extension `.mjs` to distinguish their modular JavaScript files from their regular, non-modular JavaScript files with the traditional `.js` extension. For the purposes of web browsers and `<script>` tags, the file extension is actually irrelevant. (The MIME type is relevant, however, so if you use `.mjs` files, you may need to configure your web server to serve them with the same MIME type as `.js` files.) Node's support for ES6 does use the filename extension as a hint to distinguish which module system is used by each file it loads. So if you are writing ES6 modules and want them to be usable with Node, then it may be helpful to adopt the `.mjs` naming convention.

10.3.6 Dynamic Imports with import()

We've seen that the ES6 `import` and `export` directives are completely static and enable JavaScript interpreters and other JavaScript tools to determine the relationships between modules with simple text analysis while the modules are being loaded without having to actually execute any of the code in the modules. With statically imported modules, you are guaranteed that the values you import into a module will be ready for use before any of the code in your module begins to run.

On the web, code has to be transferred over a network instead of being read from the filesystem. And once transfered, that code is often executed on mobile devices with relatively slow CPUs. This is not the kind of environment where static module imports—which require an entire program to be loaded before any of it runs—make a lot of sense.

It is common for web applications to initially load only enough of their code to render the first page displayed to the user. Then, once the user has some preliminary content to interact with, they can begin to load the often much larger amount of code needed for the rest of the web app. Web browsers make it easy to dynamically load code by using the DOM API to inject a new `<script>` tag into the current HTML document, and web apps have been doing this for many years.

Although dynamic loading has been possible for a long time, it has not been part of the language itself. That changes with the introduction of `import()` in ES2020 (as of early 2020, dynamic import is supported by all browsers that support ES6 modules). You pass a module specifier to `import()` and it returns a Promise object that represents the asynchronous process of loading and running the specified module. When the dynamic import is complete, the Promise is "fulfilled" (see Chapter 13 for complete details on asynchronous programming and Promises) and produces an object like the one you would get with the `import * as` form of the static import statement.

So instead of importing the "./stats.js" module statically, like this:

```
import * as stats from "./stats.js";
```

we might import it and use it dynamically, like this:

```
import("./stats.js").then(stats => {
    let average = stats.mean(data);
})
```

Or, in an `async` function (again, you may need to read Chapter 13 before you'll understand this code), we can simplify the code with `await`:

```
async analyzeData(data) {
    let stats = await import("./stats.js");
    return {
        average: stats.mean(data),
        stddev: stats.stddev(data)
    };
}
```

The argument to `import()` should be a module specifier, exactly like one you'd use with a static `import` directive. But with `import()`, you are not constrained to use a constant string literal: any expression that evaluates to a string in the proper form will do.

Dynamic `import()` looks like a function invocation, but it actually is not. Instead, `import()` is an operator and the parentheses are a required part of the operator syntax. The reason for this unusual bit of syntax is that `import()` needs to be able to resolve module specifiers as URLs relative to the currently running module, and this requires a bit of implementation magic that would not be legal to put in a JavaScript function. The function versus operator distinction rarely makes a difference in practice, but you'll notice it if you try writing code like `console.log(import);` or `let require = import;`.

Finally, note that dynamic `import()` is not just for web browsers. Code-packaging tools like webpack can also make good use of it. The most straightforward way to use a code bundler is to tell it the main entry point for your program and let it find all the static `import` directives and assemble everything into one large file. By strategically using dynamic `import()` calls, however, you can break that one monolithic bundle up into a set of smaller bundles that can be loaded on demand.

10.3.7 import.meta.url

There is one final feature of the ES6 module system to discuss. Within an ES6 module (but not within a regular `<script>` or a Node module loaded with `require()`), the special syntax `import.meta` refers to an object that contains metadata about the currently executing module. The `url` property of this object is the URL from which the module was loaded. (In Node, this will be a `file://` URL.)

The primary use case of `import.meta.url` is to be able to refer to images, data files, or other resources that are stored in the same directory as (or relative to) the module. The `URL()` constructor makes it easy to resolve a relative URL against an absolute

URL like `import.meta.url`. Suppose, for example, that you have written a module that includes strings that need to be localized and that the localization files are stored in an `l10n/` directory, which is in the same directory as the module itself. Your module could load its strings using a URL created with a function, like this:

```
function localStringsURL(locale) {
    return new URL(`l10n/${locale}.json`, import.meta.url);
}
```

10.4 Summary

The goal of modularity is to allow programmers to hide the implementation details of their code so that chunks of code from various sources can be assembled into large programs without worrying that one chunk will overwrite functions or variables of another. This chapter has explained three different JavaScript module systems:

- In the early days of JavaScript, modularity could only be achieved through the clever use of immediately invoked function expressions.
- Node added its own module system on top of the JavaScript language. Node modules are imported with `require()` and define their exports by setting properties of the Exports object, or by setting the `module.exports` property.
- In ES6, JavaScript finally got its own module system with `import` and `export` keywords, and ES2020 is adding support for dynamic imports with `import()`.

CHAPTER 11

The JavaScript Standard Library

Some datatypes, such as numbers and strings (Chapter 3), objects (Chapter 6), and arrays (Chapter 7) are so fundamental to JavaScript that we can consider them to be part of the language itself. This chapter covers other important but less fundamental APIs that can be thought of as defining the "standard library" for JavaScript: these are useful classes and functions that are built in to JavaScript and available to all JavaScript programs in both web browsers and in Node.[1]

The sections of this chapter are independent of one another, and you can read them in any order. They cover:

- The Set and Map classes for representing sets of values and mappings from one set of values to another set of values.
- Array-like objects known as TypedArrays that represent arrays of binary data, along with a related class for extracting values from non-array binary data.
- Regular expressions and the RegExp class, which define textual patterns and are useful for text processing. This section also covers regular expression syntax in detail.
- The Date class for representing and manipulating dates and times.
- The Error class and its various subclasses, instances of which are thrown when errors occur in JavaScript programs.

1 Not everything documented here is defined by the JavaScript language specification: some of the classes and functions documented here were first implemented in web browsers and then adopted by Node, making them de facto members of the JavaScript standard library.

- The JSON object, whose methods support serialization and deserialization of JavaScript data structures composed of objects, arrays, strings, numbers, and booleans.
- The Intl object and the classes it defines that can help you localize your JavaScript programs.
- The Console object, whose methods output strings in ways that are particularly useful for debugging programs and logging the behavior of those programs.
- The URL class, which simplifies the task of parsing and manipulating URLs. This section also covers global functions for encoding and decoding URLs and their component parts.
- `setTimeout()` and related functions for specifying code to be executed after a specified interval of time has elapsed.

Some of the sections in this chapter—notably, the sections on typed arrays and regular expressions—are quite long because there is significant background information you need to understand before you can use those types effectively. Many of the other sections, however, are short: they simply introduce a new API and show some examples of its use.

11.1 Sets and Maps

JavaScript's Object type is a versatile data structure that can be used to map strings (the object's property names) to arbitrary values. And when the value being mapped to is something fixed like `true`, then the object is effectively a set of strings.

Objects are actually used as maps and sets fairly routinely in JavaScript programming, but this is limited by the restriction to strings and complicated by the fact that objects normally inherit properties with names like "toString", which are not typically intended to be part of the map or set.

For this reason, ES6 introduces true Set and Map classes, which we'll cover in the subsections that follow.

11.1.1 The Set Class

A set is a collection of values, like an array is. Unlike arrays, however, sets are not ordered or indexed, and they do not allow duplicates: a value is either a member of a set or it is not a member; it is not possible to ask how many times a value appears in a set.

Create a Set object with the `Set()` constructor:

```
let s = new Set();       // A new, empty set
let t = new Set([1, s]); // A new set with two members
```

The argument to the `Set()` constructor need not be an array: any iterable object (including other Set objects) is allowed:

```
let t = new Set(s);                 // A new set that copies the elements of s.
let unique = new Set("Mississippi"); // 4 elements: "M", "i", "s", and "p"
```

The `size` property of a set is like the `length` property of an array: it tells you how many values the set contains:

```
unique.size        // => 4
```

Sets don't need to be initialized when you create them. You can add and remove elements at any time with `add()`, `delete()`, and `clear()`. Remember that sets cannot contain duplicates, so adding a value to a set when it already contains that value has no effect:

```
let s = new Set();  // Start empty
s.size              // => 0
s.add(1);           // Add a number
s.size              // => 1; now the set has one member
s.add(1);           // Add the same number again
s.size              // => 1; the size does not change
s.add(true);        // Add another value; note that it is fine to mix types
s.size              // => 2
s.add([1,2,3]);     // Add an array value
s.size              // => 3; the array was added, not its elements
s.delete(1)         // => true: successfully deleted element 1
s.size              // => 2: the size is back down to 2
s.delete("test")    // => false: "test" was not a member, deletion failed
s.delete(true)      // => true: delete succeeded
s.delete([1,2,3])   // => false: the array in the set is different
s.size              // => 1: there is still that one array in the set
s.clear();          // Remove everything from the set
s.size              // => 0
```

There are a few important points to note about this code:

- The `add()` method takes a single argument; if you pass an array, it adds the array itself to the set, not the individual array elements. `add()` always returns the set it is invoked on, however, so if you want to add multiple values to a set, you can used chained method calls like `s.add('a').add('b').add('c');`.
- The `delete()` method also only deletes a single set element at a time. Unlike `add()`, however, `delete()` returns a boolean value. If the value you specify was actually a member of the set, then `delete()` removes it and returns `true`. Otherwise, it does nothing and returns `false`.
- Finally, it is very important to understand that set membership is based on strict equality checks, like the `===` operator performs. A set can contain both the number `1` and the string `"1"`, because it considers them to be distinct values. When

the values are objects (or arrays or functions), they are also compared as if with ===. This is why we were unable to delete the array element from the set in this code. We added an array to the set and then tried to remove that array by passing a *different* array (albeit with the same elements) to the `delete()` method. In order for this to work, we would have had to pass a reference to exactly the same array.

Python programmers take note: this is a significant difference between JavaScript and Python sets. Python sets compare members for equality, not identity, but the trade-off is that Python sets only allow immutable members, like tuples, and do not allow lists and dicts to be added to sets.

In practice, the most important thing we do with sets is not to add and remove elements from them, but to check to see whether a specified value is a member of the set. We do this with the `has()` method:

```
let oneDigitPrimes = new Set([2,3,5,7]);
oneDigitPrimes.has(2)    // => true: 2 is a one-digit prime number
oneDigitPrimes.has(3)    // => true: so is 3
oneDigitPrimes.has(4)    // => false: 4 is not a prime
oneDigitPrimes.has("5")  // => false: "5" is not even a number
```

The most important thing to understand about sets is that they are optimized for membership testing, and no matter how many members the set has, the `has()` method will be very fast. The `includes()` method of an array also performs membership testing, but the time it takes is proportional to the size of the array, and using an array as a set can be much, much slower than using a real Set object.

The Set class is iterable, which means that you can use a `for/of` loop to enumerate all of the elements of a set:

```
let sum = 0;
for(let p of oneDigitPrimes) { // Loop through the one-digit primes
    sum += p;                  // and add them up
}
sum                            // => 17: 2 + 3 + 5 + 7
```

Because Set objects are iterable, you can convert them to arrays and argument lists with the `...` spread operator:

```
[...oneDigitPrimes]         // => [2,3,5,7]: the set converted to an Array
Math.max(...oneDigitPrimes) // => 7: set elements passed as function arguments
```

Sets are often described as "unordered collections." This isn't exactly true for the JavaScript Set class, however. A JavaScript set is unindexed: you can't ask for the first or third element of a set the way you can with an array. But the JavaScript Set class

always remembers the order that elements were inserted in, and it always uses this order when you iterate a set: the first element inserted will be the first one iterated (assuming you haven't deleted it first), and the most recently inserted element will be the last one iterated.[2]

In addition to being iterable, the Set class also implements a `forEach()` method that is similar to the array method of the same name:

```
let product = 1;
oneDigitPrimes.forEach(n => { product *= n; });
product     // => 210: 2 * 3 * 5 * 7
```

The `forEach()` of an array passes array indexes as the second argument to the function you specify. Sets don't have indexes, so the Set class's version of this method simply passes the element value as both the first and second argument.

11.1.2 The Map Class

A Map object represents a set of values known as *keys*, where each key has another value associated with (or "mapped to") it. In a sense, a map is like an array, but instead of using a set of sequential integers as the keys, maps allow us to use arbitrary values as "indexes." Like arrays, maps are fast: looking up the value associated with a key will be fast (though not as fast as indexing an array) no matter how large the map is.

Create a new map with the `Map()` constructor:

```
let m = new Map();  // Create a new, empty map
let n = new Map([   // A new map initialized with string keys mapped to numbers
    ["one", 1],
    ["two", 2]
]);
```

The optional argument to the `Map()` constructor should be an iterable object that yields two element `[key, value]` arrays. In practice, this means that if you want to initialize a map when you create it, you'll typically write out the desired keys and associated values as an array of arrays. But you can also use the `Map()` constructor to copy other maps or to copy the property names and values from an existing object:

```
let copy = new Map(n); // A new map with the same keys and values as map n
let o = { x: 1, y: 2}; // An object with two properties
let p = new Map(Object.entries(o)); // Same as new map([["x", 1], ["y", 2]])
```

Once you have created a Map object, you can query the value associated with a given key with `get()` and can add a new key/value pair with `set()`. Remember, though,

2 This predictable iteration order is another thing about JavaScript sets that Python programmers may find surprising.

that a map is a set of keys, each of which has an associated value. This is not quite the same as a set of key/value pairs. If you call `set()` with a key that already exists in the map, you will change the value associated with that key, not add a new key/value mapping. In addition to `get()` and `set()`, the Map class also defines methods that are like Set methods: use `has()` to check whether a map includes the specified key; use `delete()` to remove a key (and its associated value) from the map; use `clear()` to remove all key/value pairs from the map; and use the `size` property to find out how many keys a map contains.

```
let m = new Map();   // Start with an empty map
m.size               // => 0: empty maps have no keys
m.set("one", 1);     // Map the key "one" to the value 1
m.set("two", 2);     // And the key "two" to the value 2.
m.size               // => 2: the map now has two keys
m.get("two")         // => 2: return the value associated with key "two"
m.get("three")       // => undefined: this key is not in the set
m.set("one", true);  // Change the value associated with an existing key
m.size               // => 2: the size doesn't change
m.has("one")         // => true: the map has a key "one"
m.has(true)          // => false: the map does not have a key true
m.delete("one")      // => true: the key existed and deletion succeeded
m.size               // => 1
m.delete("three")    // => false: failed to delete a nonexistent key
m.clear();           // Remove all keys and values from the map
```

Like the `add()` method of Set, the `set()` method of Map can be chained, which allows maps to be initialized without using arrays of arrays:

```
let m = new Map().set("one", 1).set("two", 2).set("three", 3);
m.size        // => 3
m.get("two")  // => 2
```

As with Set, any JavaScript value can be used as a key or a value in a Map. This includes `null`, `undefined`, and `NaN`, as well as reference types like objects and arrays. And as with the Set class, Map compares keys by identity, not by equality, so if you use an object or array as a key, it will be considered different from every other object and array, even those with exactly the same properties or elements:

```
let m = new Map();   // Start with an empty map.
m.set({}, 1);        // Map one empty object to the number 1.
m.set({}, 2);        // Map a different empty object to the number 2.
m.size               // => 2: there are two keys in this map
m.get({})            // => undefined: but this empty object is not a key
m.set(m, undefined); // Map the map itself to the value undefined.
m.has(m)             // => true: m is a key in itself
m.get(m)             // => undefined: same value we'd get if m wasn't a key
```

Map objects are iterable, and each iterated value is a two-element array where the first element is a key and the second element is the value associated with that key. If you use the spread operator with a Map object, you'll get an array of arrays like the ones

that we passed to the `Map()` constructor. And when iterating a map with a `for/of` loop, it is idiomatic to use destructuring assignment to assign the key and value to separate variables:

```
let m = new Map([["x", 1], ["y", 2]]);
[...m]    // => [["x", 1], ["y", 2]]

for(let [key, value] of m) {
    // On the first iteration, key will be "x" and value will be 1
    // On the second iteration, key will be "y" and value will be 2
}
```

Like the Set class, the Map class iterates in insertion order. The first key/value pair iterated will be the one least recently added to the map, and the last pair iterated will be the one most recently added.

If you want to iterate just the keys or just the associated values of a map, use the `keys()` and `values()` methods: these return iterable objects that iterate keys and values, in insertion order. (The `entries()` method returns an iterable object that iterates key/value pairs, but this is exactly the same as iterating the map directly.)

```
[...m.keys()]     // => ["x", "y"]: just the keys
[...m.values()]   // => [1, 2]: just the values
[...m.entries()]  // => [["x", 1], ["y", 2]]: same as [...m]
```

Map objects can also be iterated using the `forEach()` method that was first implemented by the Array class.

```
m.forEach((value, key) => {  // note value, key NOT key, value
    // On the first invocation, value will be 1 and key will be "x"
    // On the second invocation, value will be 2 and key will be "y"
});
```

It may seem strange that the value parameter comes before the key parameter in the code above, since with `for/of` iteration, the key comes first. As noted at the start of this section, you can think of a map as a generalized array in which integer array indexes are replaced with arbitrary key values. The `forEach()` method of arrays passes the array element first and the array index second, so, by analogy, the `forEach()` method of a map passes the map value first and the map key second.

11.1.3 WeakMap and WeakSet

The WeakMap class is a variant (but not an actual subclass) of the Map class that does not prevent its key values from being garbage collected. Garbage collection is the process by which the JavaScript interpreter reclaims the memory of objects that are no longer "reachable" and cannot be used by the program. A regular map holds "strong" references to its key values, and they remain reachable through the map, even if all other references to them are gone. The WeakMap, by contrast, keeps "weak"

references to its key values so that they are not reachable through the WeakMap, and their presence in the map does not prevent their memory from being reclaimed.

The `WeakMap()` constructor is just like the `Map()` constructor, but there are some significant differences between WeakMap and Map:

- WeakMap keys must be objects or arrays; primitive values are not subject to garbage collection and cannot be used as keys.
- WeakMap implements only the `get()`, `set()`, `has()`, and `delete()` methods. In particular, WeakMap is not iterable and does not define `keys()`, `values()`, or `forEach()`. If WeakMap was iterable, then its keys would be reachable and it wouldn't be weak.
- Similarly, WeakMap does not implement the `size` property because the size of a WeakMap could change at any time as objects are garbage collected.

The intended use of WeakMap is to allow you to associate values with objects without causing memory leaks. Suppose, for example, that you are writing a function that takes an object argument and needs to perform some time-consuming computation on that object. For efficiency, you'd like to cache the computed value for later reuse. If you use a Map object to implement the cache, you will prevent any of the objects from ever being reclaimed, but by using a WeakMap, you avoid this problem. (You can often achieve a similar result using a private Symbol property to cache the computed value directly on the object. See §6.10.3.)

WeakSet implements a set of objects that does not prevent those objects from being garbage collected. The `WeakSet()` constructor works like the `Set()` constructor, but WeakSet objects differ from Set objects in the same ways that WeakMap objects differ from Map objects:

- WeakSet does not allow primitive values as members.
- WeakSet implements only the `add()`, `has()`, and `delete()` methods and is not iterable.
- WeakSet does not have a `size` property.

WeakSet is not frequently used: its use cases are like those for WeakMap. If you want to mark (or "brand") an object as having some special property or type, for example, you could add it to a WeakSet. Then, elsewhere, when you want to check for that property or type, you can test for membership in that WeakSet. Doing this with a regular set would prevent all marked objects from being garbage collected, but this is not a concern when using WeakSet.

11.2 Typed Arrays and Binary Data

Regular JavaScript arrays can have elements of any type and can grow or shrink dynamically. JavaScript implementations perform lots of optimizations so that typical uses of JavaScript arrays are very fast. Nevertheless, they are still quite different from the array types of lower-level languages like C and Java. *Typed arrays*, which are new in ES6,[3] are much closer to the low-level arrays of those languages. Typed arrays are not technically arrays (`Array.isArray()` returns `false` for them), but they implement all of the array methods described in §7.8 plus a few more of their own. They differ from regular arrays in some very important ways, however:

- The elements of a typed array are all numbers. Unlike regular JavaScript numbers, however, typed arrays allow you to specify the type (signed and unsigned integers and IEEE-754 floating point) and size (8 bits to 64 bits) of the numbers to be stored in the array.
- You must specify the length of a typed array when you create it, and that length can never change.
- The elements of a typed array are always initialized to 0 when the array is created.

11.2.1 Typed Array Types

JavaScript does not define a TypedArray class. Instead, there are 11 kinds of typed arrays, each with a different element type and constructor:

Constructor	Numeric type
`Int8Array()`	signed bytes
`Uint8Array()`	unsigned bytes
`Uint8ClampedArray()`	unsigned bytes without rollover
`Int16Array()`	signed 16-bit short integers
`Uint16Array()`	unsigned 16-bit short integers
`Int32Array()`	signed 32-bit integers
`Uint32Array()`	unsigned 32-bit integers
`BigInt64Array()`	signed 64-bit BigInt values (ES2020)
`BigUint64Array()`	unsigned 64-bit BigInt values (ES2020)
`Float32Array()`	32-bit floating-point value
`Float64Array()`	64-bit floating-point value: a regular JavaScript number

3 Typed arrays were first introduced to client-side JavaScript when web browsers added support for WebGL graphics. What is new in ES6 is that they have been elevated to a core language feature.

The types whose names begin with `Int` hold signed integers, of 1, 2, or 4 bytes (8, 16, or 32 bits). The types whose names begin with `Uint` hold unsigned integers of those same lengths. The "BigInt" and "BigUint" types hold 64-bit integers, represented in JavaScript as BigInt values (see §3.2.5). The types that begin with `Float` hold floating-point numbers. The elements of a `Float64Array` are of the same type as regular JavaScript numbers. The elements of a `Float32Array` have lower precision and a smaller range but require only half the memory. (This type is called `float` in C and Java.)

`Uint8ClampedArray` is a special-case variant on `Uint8Array`. Both of these types hold unsigned bytes and can represent numbers between 0 and 255. With `Uint8Array`, if you store a value larger than 255 or less than zero into an array element, it "wraps around," and you get some other value. This is how computer memory works at a low level, so this is very fast. `Uint8ClampedArray` does some extra type checking so that, if you store a value greater than 255 or less than 0, it "clamps" to 255 or 0 and does not wrap around. (This clamping behavior is required by the HTML `<canvas>` element's low-level API for manipulating pixel colors.)

Each of the typed array constructors has a `BYTES_PER_ELEMENT` property with the value 1, 2, 4, or 8, depending on the type.

11.2.2 Creating Typed Arrays

The simplest way to create a typed array is to call the appropriate constructor with one numeric argument that specifies the number of elements you want in the array:

```
let bytes = new Uint8Array(1024);     // 1024 bytes
let matrix = new Float64Array(9);     // A 3x3 matrix
let point = new Int16Array(3);        // A point in 3D space
let rgba = new Uint8ClampedArray(4); // A 4-byte RGBA pixel value
let sudoku = new Int8Array(81);       // A 9x9 sudoku board
```

When you create typed arrays in this way, the array elements are all guaranteed to be initialized to `0`, `0n`, or `0.0`. But if you know the values you want in your typed array, you can also specify those values when you create the array. Each of the typed array constructors has static `from()` and `of()` factory methods that work like `Array.from()` and `Array.of()`:

```
let white = Uint8ClampedArray.of(255, 255, 255, 0);  // RGBA opaque white
```

Recall that the `Array.from()` factory method expects an array-like or iterable object as its first argument. The same is true for the typed array variants, except that the iterable or array-like object must also have numeric elements. Strings are iterable, for example, but it would make no sense to pass them to the `from()` factory method of a typed array.

If you are just using the one-argument version of `from()`, you can drop the `.from` and pass your iterable or array-like object directly to the constructor function, which

behaves exactly the same. Note that both the constructor and the `from()` factory method allow you to copy existing typed arrays, while possibly changing the type:

```
let ints = Uint32Array.from(white);  // The same 4 numbers, but as ints
```

When you create a new typed array from an existing array, iterable, or array-like object, the values may be truncated in order to fit the type constraints of your array. There are no warnings or errors when this happens:

```
// Floats truncated to ints, longer ints truncated to 8 bits
Uint8Array.of(1.23, 2.99, 45000) // => new Uint8Array([1, 2, 200])
```

Finally, there is one more way to create typed arrays that involves the ArrayBuffer type. An ArrayBuffer is an opaque reference to a chunk of memory. You can create one with the constructor; just pass in the number of bytes of memory you'd like to allocate:

```
let buffer = new ArrayBuffer(1024*1024);
buffer.byteLength   // => 1024*1024; one megabyte of memory
```

The ArrayBuffer class does not allow you to read or write any of the bytes that you have allocated. But you can create typed arrays that use the buffer's memory and that do allow you to read and write that memory. To do this, call the typed array constructor with an ArrayBuffer as the first argument, a byte offset within the array buffer as the second argument, and the array length (in elements, not in bytes) as the third argument. The second and third arguments are optional. If you omit both, then the array will use all of the memory in the array buffer. If you omit only the length argument, then your array will use all of the available memory between the start position and the end of the array. One more thing to bear in mind about this form of the typed array constructor: arrays must be memory aligned, so if you specify a byte offset, the value should be a multiple of the size of your type. The `Int32Array()` constructor requires a multiple of four, for example, and the `Float64Array()` requires a multiple of eight.

Given the ArrayBuffer created earlier, you could create typed arrays like these:

```
let asbytes = new Uint8Array(buffer);          // Viewed as bytes
let asints = new Int32Array(buffer);           // Viewed as 32-bit signed ints
let lastK = new Uint8Array(buffer, 1023*1024); // Last kilobyte as bytes
let ints2 = new Int32Array(buffer, 1024, 256); // 2nd kilobyte as 256 integers
```

These four typed arrays offer four different views into the memory represented by the ArrayBuffer. It is important to understand that all typed arrays have an underlying ArrayBuffer, even if you do not explicitly specify one. If you call a typed array constructor without passing a buffer object, a buffer of the appropriate size will be automatically created. As described later, the `buffer` property of any typed array refers to its underlying ArrayBuffer object. The reason to work directly with ArrayBuffer objects is that sometimes you may want to have multiple typed array views of a single buffer.

11.2.3 Using Typed Arrays

Once you have created a typed array, you can read and write its elements with regular square-bracket notation, just as you would with any other array-like object:

```
// Return the largest prime smaller than n, using the sieve of Eratosthenes
function sieve(n) {
    let a = new Uint8Array(n+1);         // a[x] will be 1 if x is composite
    let max = Math.floor(Math.sqrt(n));  // Don't do factors higher than this
    let p = 2;                           // 2 is the first prime
    while(p <= max) {                    // For primes less than max
        for(let i = 2*p; i <= n; i += p) // Mark multiples of p as composite
            a[i] = 1;
        while(a[++p]) /* empty */;       // The next unmarked index is prime
    }
    while(a[n]) n--;                     // Loop backward to find the last prime
    return n;                            // And return it
}
```

The function here computes the largest prime number smaller than the number you specify. The code is exactly the same as it would be with a regular JavaScript array, but using `Uint8Array()` instead of `Array()` makes the code run more than four times faster and use eight times less memory in my testing.

Typed arrays are not true arrays, but they re-implement most array methods, so you can use them pretty much just like you'd use regular arrays:

```
let ints = new Int16Array(10);       // 10 short integers
ints.fill(3).map(x=>x*x).join("")    // => "9999999999"
```

Remember that typed arrays have fixed lengths, so the `length` property is read-only, and methods that change the length of the array (such as `push()`, `pop()`, `unshift()`, `shift()`, and `splice()`) are not implemented for typed arrays. Methods that alter the contents of an array without changing the length (such as `sort()`, `reverse()`, and `fill()`) are implemented. Methods like `map()` and `slice()` that return new arrays return a typed array of the same type as the one they are called on.

11.2.4 Typed Array Methods and Properties

In addition to standard array methods, typed arrays also implement a few methods of their own. The `set()` method sets multiple elements of a typed array at once by copying the elements of a regular or typed array into a typed array:

```
let bytes = new Uint8Array(1024);        // A 1K buffer
let pattern = new Uint8Array([0,1,2,3]); // An array of 4 bytes
bytes.set(pattern);      // Copy them to the start of another byte array
bytes.set(pattern, 4);   // Copy them again at a different offset
bytes.set([0,1,2,3], 8); // Or just copy values direct from a regular array
bytes.slice(0, 12)       // => new Uint8Array([0,1,2,3,0,1,2,3,0,1,2,3])
```

The `set()` method takes an array or typed array as its first argument and an element offset as its optional second argument, which defaults to 0 if left unspecified. If you are copying values from one typed array to another, the operation will likely be extremely fast.

Typed arrays also have a `subarray` method that returns a portion of the array on which it is called:

```
let ints = new Int16Array([0,1,2,3,4,5,6,7,8,9]);       // 10 short integers
let last3 = ints.subarray(ints.length-3, ints.length);  // Last 3 of them
last3[0]       // => 7: this is the same as ints[7]
```

`subarray()` takes the same arguments as the `slice()` method and seems to work the same way. But there is an important difference. `slice()` returns the specified elements in a new, independent typed array that does not share memory with the original array. `subarray()` does not copy any memory; it just returns a new view of the same underlying values:

```
ints[9] = -1;  // Change a value in the original array and...
last3[2]       // => -1: it also changes in the subarray
```

The fact that the `subarray()` method returns a new view of an existing array brings us back to the topic of ArrayBuffers. Every typed array has three properties that relate to the underlying buffer:

```
last3.buffer                // The ArrayBuffer object for a typed array
last3.buffer === ints.buffer // => true: both are views of the same buffer
last3.byteOffset            // => 14: this view starts at byte 14 of the buffer
last3.byteLength            // => 6: this view is 6 bytes (3 16-bit ints) long
last3.buffer.byteLength     // => 20: but the underlying buffer has 20 bytes
```

The `buffer` property is the ArrayBuffer of the array. `byteOffset` is the starting position of the array's data within the underlying buffer. And `byteLength` is the length of the array's data in bytes. For any typed array, `a`, this invariant should always be true:

```
a.length * a.BYTES_PER_ELEMENT === a.byteLength  // => true
```

ArrayBuffers are just opaque chunks of bytes. You can access those bytes with typed arrays, but an ArrayBuffer is not itself a typed array. Be careful, however: you can use numeric array indexing with ArrayBuffers just as you can with any JavaScript object. Doing so does not give you access to the bytes in the buffer, but it can cause confusing bugs:

```
let bytes = new Uint8Array(8);
bytes[0] = 1;           // Set the first byte to 1
bytes.buffer[0]         // => undefined: buffer doesn't have index 0
bytes.buffer[1] = 255;  // Try incorrectly to set a byte in the buffer
bytes.buffer[1]         // => 255: this just sets a regular JS property
bytes[1]                // => 0: the line above did not set the byte
```

We saw previously that you can create an ArrayBuffer with the `ArrayBuffer()` constructor and then create typed arrays that use that buffer. Another approach is to create an initial typed array, then use the buffer of that array to create other views:

```
let bytes = new Uint8Array(1024);            // 1024 bytes
let ints = new Uint32Array(bytes.buffer);    // or 256 integers
let floats = new Float64Array(bytes.buffer); // or 128 doubles
```

11.2.5 DataView and Endianness

Typed arrays allow you to view the same sequence of bytes in chunks of 8, 16, 32, or 64 bits. This exposes the "endianness": the order in which bytes are arranged into longer words. For efficiency, typed arrays use the native endianness of the underlying hardware. On little-endian systems, the bytes of a number are arranged in an ArrayBuffer from least significant to most significant. On big-endian platforms, the bytes are arranged from most significant to least significant. You can determine the endianness of the underlying platform with code like this:

```
// If the integer 0x00000001 is arranged in memory as 01 00 00 00, then
// we're on a little-endian platform. On a big-endian platform, we'd get
// bytes 00 00 00 01 instead.
let littleEndian = new Int8Array(new Int32Array([1]).buffer)[0] === 1;
```

Today, the most common CPU architectures are little-endian. Many network protocols, and some binary file formats, require big-endian byte ordering, however. If you're using typed arrays with data that came from the network or from a file, you can't just assume that the platform endianness matches the byte order of the data. In general, when working with external data, you can use Int8Array and Uint8Array to view the data as an array of individual bytes, but you should not use the other typed arrays with multibyte word sizes. Instead, you can use the DataView class, which defines methods for reading and writing values from an ArrayBuffer with explicitly specified byte ordering:

```
// Assume we have a typed array of bytes of binary data to process. First,
// we create a DataView object so we can flexibly read and write
// values from those bytes
let view = new DataView(bytes.buffer,
                        bytes.byteOffset,
                        bytes.byteLength);

let int = view.getInt32(0);     // Read big-endian signed int from byte 0
int = view.getInt32(4, false);  // Next int is also big-endian
int = view.getUint32(8, true);  // Next int is little-endian and unsigned
view.setUint32(8, int, false);  // Write it back in big-endian format
```

DataView defines 10 `get` methods for each of the 10 typed array classes (excluding Uint8ClampedArray). They have names like `getInt16()`, `getUint32()`, `getBigInt64()`, and `getFloat64()`. The first argument is the byte offset within the ArrayBuffer at which the value begins. All of these getter methods, other than

`getInt8()` and `getUint8()`, accept an optional boolean value as their second argument. If the second argument is omitted or is `false`, big-endian byte ordering is used. If the second argument is `true`, little-endian ordering is used.

DataView also defines 10 corresponding Set methods that write values into the underlying ArrayBuffer. The first argument is the offset at which the value begins. The second argument is the value to write. Each of the methods, except `setInt8()` and `setUint8()`, accepts an optional third argument. If the argument is omitted or is `false`, the value is written in big-endian format with the most significant byte first. If the argument is `true`, the value is written in little-endian format with the least significant byte first.

Typed arrays and the DataView class give you all the tools you need to process binary data and enable you to write JavaScript programs that do things like decompressing ZIP files or extracting metadata from JPEG files.

11.3 Pattern Matching with Regular Expressions

A *regular expression* is an object that describes a textual pattern. 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. In order to use the RegExp API effectively, however, you must also learn how to describe patterns of text using the regular expression grammar, which is essentially a mini programming language of its own. Fortunately, the JavaScript regular expression grammar is quite similar to the grammar used by many other programming languages, so you may already be familiar with it. (And if you are not, the effort you invest in learning JavaScript regular expressions will probably be useful to you in other programming contexts as well.)

The subsections that follow describe the regular expression grammar first, and then, after explaining how to write regular expressions, they explain how you can use them with methods of the String and RegExp classes.

11.3.1 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:

```
let 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:

```
let pattern = new RegExp("s$");
```

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 meta-character that matches the end of a string. Thus, this regular expression matches any string that contains the letter "s" as its last character.

As we'll see, regular expressions can also have one or more flag characters that affect how they work. Flags are specified following the second slash character in RegExp literals, or as a second string argument to the `RegExp()` constructor. If we wanted to match strings that end with "s" or "S", for example, we could use the `i` flag with our regular expression to indicate that we want case-insensitive matching:

```
let pattern = /s$/i;
```

The following sections describe the various characters and meta-characters used in JavaScript regular expressions.

Literal characters

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 11-1 lists these characters.

Table 11-1. 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`)
`\r`	Carriage return (`\u000D`)
`\x`*nn*	The Latin character specified by the hexadecimal number *nn*; for example, `\x0A` is the same as `\n`.

Character	Matches
\u*xxxx*	The Unicode character specified by the hexadecimal number *xxxx*; for example, \u0009 is the same as \t.
\u{*n*}	The Unicode character specified by the codepoint *n*, where *n* is one to six hexadecimal digits between 0 and 10FFFF. Note that this syntax is only supported in regular expressions that use the u flag.
\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: /\\/. (And if you use the RegExp() constructor, keep in mind that any backslashes in your regular expression need to be doubled, since strings also use backslashes as an escape character.)

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]/. (And if you want to include an actual hyphen in your character class, simply make it the last character before the right bracket.)

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 11-2 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 11-2. 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. Or, if the RegExp uses the `s` flag, then a period matches any character, including line terminators.
`\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.
`\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]/`.

Unicode Character Classes

In ES2018, if a regular expression uses the `u` flag, then character classes `\p{...}` and its negation `\P{...}` are supported. (As of early 2020, this is implemented by Node, Chrome, Edge, and Safari, but not Firefox.) These character classes are based on properties defined by the Unicode standard, and the set of characters they represent may change as Unicode evolves.

The `\d` character class matches only ASCII digits. If you want to match one decimal digit from any of the world's writing systems, you can use `/\p{Decimal_Number}/u`.

And if you want to match any one character that is *not* a decimal digit in any language, you can capitalize the `p` and write `\P{Decimal_Number}`. If you want to match any number-like character, including fractions and roman numerals, you can use `\p{Number}`. Note that "Decimal_Number" and "Number" are not specific to JavaScript or to regular expression grammar: it is the name of a category of characters defined by the Unicode standard.

The `\w` character class only works for ASCII text, but with `\p`, we can approximate an internationalized version like this:

```
/[\p{Alphabetic}\p{Decimal_Number}\p{Mark}]/u
```

(Though to be fully compatible with the complexity of the world's languages, we really need to add in the categories "Connector_Punctuation" and "Join_Control" as well.)

As a final example, the `\p` syntax also allows us to define regular expressions that match characters from a particular alphabet or script:

```
let greekLetter = /\p{Script=Greek}/u;
let cyrillicLetter = /\p{Script=Cyrillic}/u;
```

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 11-3 summarizes the repetition syntax.

Table 11-3. 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:

```
let r = /\d{2,4}/; // Match between two and four digits
r = /\w{3}\d?/;    // Match exactly three word characters and an optional digit
r = /\s+java\s+/;  // Match "java" with one or more spaces before and after
r = /[^(]*/;       // Match zero or more characters that are not open parens
```

Note that in all of these examples, the repetition specifiers apply to the single character or character class that precedes them. If you want to match repetitions of more complicated expressions, you'll need to define a group with parentheses, which are explained in the following sections.

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!

Non-greedy repetition

The repetition characters listed in Table 11-3 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 non-greedy way. Simply follow the repetition character or characters with a question mark: `??`, `+?`, `*?`, or even `{1,5}?`. For example, the regular expression `/a+/` matches one or more occurrences of the letter a. When applied to the string "aaa", it matches all three letters. But `/a+?/` matches one or more occurrences of the letter a, matching as few characters as necessary. When applied to the same string, this pattern matches only the first letter a.

Using non-greedy repetition may not always produce the results you expect. Consider the pattern `/a+b/`, which matches 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 non-greedy version: `/a+?b/`. This should match the letter b preceded by the fewest number of a's possible. When applied to the same string "aaab", you might expect it to match only 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.

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 this section.) 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/`.)

When we cover the RegExp API later, 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:

```
/([Jj]ava(?:[Ss]cript)?)\sis\s(fun\w*)/
```

In this example, 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 11-4 summarizes the regular expression alternation, grouping, and referencing operators.

Table 11-4. 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.

Named Capture Groups

ES2018 standardizes a new feature that can make regular expressions more self-documenting and easier to understand. This new feature is known as "named capture groups" and it allows us to associate a name with each left parenthesis in a regular expression so that we can refer to the matching text by name rather than by number. Equally important: using names allows someone reading the code to more easily understand the purpose of that portion of the regular expression. As of early 2020, this feature is implemented in Node, Chrome, Edge, and Safari, but not yet by Firefox.

To name a group, use `(?<...>` instead of `(` and put the name between the angle brackets. For example, here is a regular expression that might be used to check the formatting of the final line of a US mailing address:

```
/(?<city>\w+) (?<state>[A-Z]{2}) (?<zipcode>\d{5})(?<zip9>-\d{4})?/
```

Notice how much context the group names provide to make the regular expression easier to understand. In §11.3.2, when we discuss the String `replace()` and `match()` methods and the RegExp `exec()` method, you'll see how the RegExp API allows you to refer to the text that matches each of these groups by name rather than by position.

If you want to refer back to a named capture group within a regular expression, you can do that by name as well. In the preceding example, we were able to use a regular expression "backreference" to write a RegExp that would match a single- or double-quoted string where the open and close quotes had to match. We could rewrite this RegExp using a named capturing group and a named backreference like this:

```
/(?<quote>['"])[^'"]*\k<quote>/
```

The `\k<quote>` is a named backreference to the named group that captures the open quotation mark.

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 an ASCII 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.[4] Elements such as `\b` do not specify any 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

4 Except within a character class (square brackets), where `\b` matches the backspace character.

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 11-5 summarizes regular expression anchors.

Table 11-5. Regular expression anchor characters

Character	Meaning
`^`	Match the beginning of the string or, with the `m` flag, the beginning of a line.
`$`	Match the end of the string and, with the `m` flag, 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*.

Lookbehind Assertions

ES2018 extends regular expression syntax to allow "lookbehind" assertions. These are like lookahead assertions but refer to text before the current match position. As of early 2020, these are implemented in Node, Chrome, and Edge, but not Firefox or Safari.

Specify a positive lookbehind assertion with `(?<=...)` and a negative lookbehind assertion with `(?<!...)`. For example, if you were working with US mailing

addresses, you could match a 5-digit zip code, but only when it follows a two-letter state abbreviation, like this:

```
/(?<= [A-Z]{2} )\d{5}/
```

And you could match a string of digits that is not preceded by a Unicode currency symbol with a negative lookbehind assertion like this:

```
/(?<![\p{Currency_Symbol}\d.])\d+(\.\d+)?/u
```

Flags

Every regular expression can have one or more flags associated with it to alter its matching behavior. JavaScript defines six possible flags, each of which is represented by a single letter. Flags are specified after the second `/` character of a regular expression literal or as a string passed as the second argument to the `RegExp()` constructor. The supported flags and their meanings are:

`g`
: The `g` flag indicates that the regular expression is "global"—that is, that we intend to use it to find all matches within a string rather than just finding the first match. This flag does not alter the way that pattern matching is done, but, as we'll see later, it does alter the behavior of the String `match()` method and the RegExp `exec()` method in important ways.

`i`
: The `i` flag specifies that pattern matching should be case-insensitive.

`m`
: The `m` flag specifies that matching should be done in "multiline" mode. It says that the RegExp will be used with multiline strings and that the `^` and `$` anchors should match both the beginning and end of the string and also the beginning and end of individual lines within the string.

`s`
: Like the `m` flag, the `s` flag is also useful when working with text that includes newlines. Normally, a "." in a regular expression matches any character except a line terminator. When the `s` flag is used, however, "." will match any character, including line terminators. The `s` flag was added to JavaScript in ES2018 and, as of early 2020, is supported in Node, Chrome, Edge, and Safari, but not Firefox.

`u`
: The `u` flag stands for Unicode, and it makes the regular expression match full Unicode codepoints rather than matching 16-bit values. This flag was introduced in ES6, and you should make a habit of using it on all regular expressions unless you have some reason not to. If you do not use this flag, then your RegExps will

not work well with text that includes emoji and other characters (including many Chinese characters) that require more than 16 bits. Without the `u` flag, the "." character matches any 1 UTF-16 16-bit value. With the flag, however, "." matches one Unicode codepoint, including those that have more than 16 bits. Setting the `u` flag on a RegExp also allows you to use the new `\u{...}` escape sequence for Unicode character and also enables the `\p{...}` notation for Unicode character classes.

`y`

The `y` flag indicates that the regular expression is "sticky" and should match at the beginning of a string or at the first character following the previous match. When used with a regular expression that is designed to find a single match, it effectively treats that regular expression as if it begins with `^` to anchor it to the beginning of the string. This flag is more useful with regular expressions that are used repeatedly to find all matches within a string. In this case, it causes special behavior of the String `match()` method and the RegExp `exec()` method to enforce that each subsequent match is anchored to the string position at which the last one ended.

These flags may be specified in any combination and in any order. For example, if you want your regular expression to be Unicode-aware to do case-insensitive matching and you intend to use it to find multiple matches within a string, you would specify the flags `uig`, `gui`, or any other permutation of these three letters.

11.3.2 String Methods for Pattern Matching

Until now, we have been describing the grammar used to define regular expressions, but not explaining how those regular expressions can actually be used in JavaScript code. We are now switching to cover the API for using RegExp objects. This section begins by explaining the string methods 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.

search()

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:

```
"JavaScript".search(/script/ui)  // => 4
"Python".search(/script/ui)      // => -1
```

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.

replace()

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 quotation marks in a string with other characters:

```
// A quote is a quotation mark, followed by any number of
// nonquotation mark characters (which we capture), followed
// by another quotation mark.
let quote = /"([^"]*)"/g;
// Replace the straight quotation marks with guillemets
// leaving the quoted text (stored in $1) unchanged.
'He said "stop"'.replace(quote, '«$1»')  // => 'He said «stop»'
```

If your RegExp uses named capture groups, then you can refer to the matching text by name rather than by number:

```
let quote = /"(?<quotedText>[^"]*)"/g;
'He said "stop"'.replace(quote, '«$<quotedText>»')  // => 'He said «stop»'
```

Instead of passing a replacement string as the second argument to `replace()`, you can also pass a function that will be invoked to compute the replacement value. The replacement function is invoked with a number of arguments. First is the entire matched text. Next, if the RegExp has capturing groups, then the substrings that were captured by those groups are passed as arguments. The next argument is the position within the string at which the match was found. After that, the entire string that `replace()` was called on is passed. And finally, if the RegExp contained any named

capture groups, the last argument to the replacement function is an object whose property names match the capture group names and whose values are the matching text. As an example, here is code that uses a replacement function to convert decimal integers in a string to hexadecimal:

```
let s = "15 times 15 is 225";
s.replace(/\d+/gu, n => parseInt(n).toString(16))  // => "f times f is e1"
```

match()

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, or `null` if no match is found. If the regular expression has the `g` flag set, the method returns an array of all matches that appear in the string. For example:

```
"7 plus 8 equals 15".match(/\d+/g)  // => ["7", "8", "15"]
```

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. In this nonglobal case, `match()` still returns an array, but the array elements are completely different. Without the `g` flag, the first element of the returned array is the matching string, and any remaining elements are the substrings matching the parenthesized capturing groups 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[1]` is the same string as `$1`, `a[2]` is the same as `$2`, and so on.

For example, consider parsing a URL[5] with the following code:

```
// A very simple URL parsing RegExp
let url = /(\w+):\/\/([\w.]+)\/(\S*)/;
let text = "Visit my blog at http://www.example.com/~david";
let match = text.match(url);
let fullurl, protocol, host, path;
if (match !== null) {
    fullurl = match[0];   // fullurl == "http://www.example.com/~david"
    protocol = match[1];  // protocol == "http"
    host = match[2];      // host == "www.example.com"
    path = match[3];      // path == "~david"
}
```

In this non-global case, the array returned by `match()` also has some object properties in addition to the numbered array elements. The `input` property refers to the string

5 Parsing URLs with regular expressions is not a good idea. See §11.9 for a more robust URL parser.

on which `match()` was called. The `index` property is the position within that string at which the match starts. And if the regular expression contains named capture groups, then the returned array also has a `groups` property whose value is an object. The properties of this object match the names of the named groups, and the values are the matching text. We could rewrite the previous URL parsing example, for example, like this:

```
let url = /(?<protocol>\w+):\/\/(?<host>[\w.]+)\/(?<path>\S*)/;
let text = "Visit my blog at http://www.example.com/~david";
let match = text.match(url);
match[0]              // => "http://www.example.com/~david"
match.input           // => text
match.index           // => 17
match.groups.protocol // => "http"
match.groups.host     // => "www.example.com"
match.groups.path     // => "~david"
```

We've seen that `match()` behaves quite differently depending on whether the RegExp has the `g` flag set or not. There are also important but less dramatic differences in behavior when the `y` flag is set. Recall that the `y` flag makes a regular expression "sticky" by constraining where in the string matches can begin. If a RegExp has both the `g` and `y` flags set, then `match()` returns an array of matched strings, just as it does when `g` is set without `y`. But the first match must begin at the start of the string, and each subsequent match must begin at the character immediately following the previous match.

If the `y` flag is set without `g`, then `match()` tries to find a single match, and, by default, this match is constrained to the start of the string. You can change this default match start position, however, by setting the `lastIndex` property of the RegExp object at the index at which you want to match at. If a match is found, then this `lastIndex` will be automatically updated to the first character after the match, so if you call `match()` again, in this case, it will look for a subsequent match. (`lastIndex` may seem like a strange name for a property that specifies the position at which to begin the *next* match. We will see it again when we cover the RegExp `exec()` method, and its name may make more sense in that context.)

```
let vowel = /[aeiou]/y;  // Sticky vowel match
"test".match(vowel)      // => null: "test" does not begin with a vowel
vowel.lastIndex = 1;     // Specify a different match position
"test".match(vowel)[0]   // => "e": we found a vowel at position 1
vowel.lastIndex          // => 2: lastIndex was automatically updated
"test".match(vowel)      // => null: no vowel at position 2
vowel.lastIndex          // => 0: lastIndex gets reset after failed match
```

It is worth noting that passing a non-global regular expression to the `match()` method of a string is the same as passing the string to the `exec()` method of the regular expression: the returned array and its properties are the same in both cases.

matchAll()

The `matchAll()` method is defined in ES2020, and as of early 2020 is implemented by modern web browsers and Node. `matchAll()` expects a RegExp with the `g` flag set. Instead of returning an array of matching substrings like `match()` does, however, it returns an iterator that yields the kind of match objects that `match()` returns when used with a non-global RegExp. This makes `matchAll()` the easiest and most general way to loop through all matches within a string.

You might use `matchAll()` to loop through the words in a string of text like this:

```
// One or more Unicode alphabetic characters between word boundaries
const words = /\b\p{Alphabetic}+\b/gu; // \p is not supported in Firefox yet
const text = "This is a naïve test of the matchAll() method.";
for(let word of text.matchAll(words)) {
    console.log(`Found '${word[0]}' at index ${word.index}.`);
}
```

You can set the `lastIndex` property of a RegExp object to tell `matchAll()` what index in the string to begin matching at. Unlike the other pattern-matching methods, however, `matchAll()` never modifies the `lastIndex` property of the RegExp you call it on, and this makes it much less likely to cause bugs in your code.

split()

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. It can be used with a string argument like this:

```
"123,456,789".split(",")           // => ["123", "456", "789"]
```

The `split()` method can also take a regular expression as its argument, and this allows you to specify more general separators. Here we call it with a separator that includes an arbitrary amount of whitespace on either side:

```
"1, 2, 3,\n4, 5".split(/\s*,\s*/)  // => ["1", "2", "3", "4", "5"]
```

Surprisingly, if you call `split()` with a RegExp delimiter and the regular expression includes capturing groups, then the text that matches the capturing groups will be included in the returned array. For example:

```
const htmlTag = /<([^>]+)>/;  // < followed by one or more non->, followed by >
"Testing<br/>1,2,3".split(htmlTag)  // => ["Testing", "br/", "1,2,3"]
```

11.3.3 The RegExp Class

This section documents the `RegExp()` constructor, the properties of RegExp instances, and two important pattern-matching methods defined by the RegExp class.

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`, `s`, `u`, `y`, or any combination of those letters.

For example:

```
// Find all five-digit numbers in a string. Note the double \\ in this case.
let 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()`.

Instead of passing a string as the first argument to `RegExp()`, you can also pass a RegExp object. This allows you to copy a regular expression and change its flags:

```
let exactMatch = /JavaScript/;
let caseInsensitive = new RegExp(exactMatch, "i");
```

RegExp properties

RegExp objects have the following properties:

`source`
: This read-only property is the source text of the regular expression: the characters that appear between the slashes in a RegExp literal.

`flags`
: This read-only property is a string that specifies the set of letters that represent the flags for the RegExp.

`global`
: A read-only boolean property that is true if the `g` flag is set.

`ignoreCase`
: A read-only boolean property that is true if the `i` flag is set.

`multiline`
: A read-only boolean property that is true if the `m` flag is set.

`dotAll`
: A read-only boolean property that is true if the `s` flag is set.

`unicode`
: A read-only boolean property that is true if the `u` flag is set.

`sticky`
: A read-only boolean property that is true if the `y` flag is set.

`lastIndex`
: This property is a read/write integer. For patterns with the `g` or `y` flags, it specifies the character position at which the next search is to begin. It is used by the `exec()` and `test()` methods, described in the next two subsections.

test()

The `test()` method of the RegExp class is the simplest way to use a regular expression. It takes a single string argument and returns `true` if the string matches the pattern or `false` if it does not match.

`test()` works by simply calling the (much more complicated) `exec()` method described in the next section and returning `true` if `exec()` returns a non-null value. Because of this, if you use `test()` with a RegExp that uses the `g` or `y` flags, then its behavior depends on the value of the `lastIndex` property of the RegExp object, which can change unexpectedly. See "The lastIndex Property and RegExp Reuse" on page 299 for more details.

exec()

The RegExp `exec()` method is the most general and powerful way to use regular expressions. It takes a single string argument and looks for a match in that string. If no match is found, it returns `null`. If a match is found, however, it returns an array just like the array returned by the `match()` method for non-global 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 capturing groups. The returned array also has named properties: the `index` property contains the character position at which the match occurred, and the `input` property specifies the string that was searched, and the `groups` property, if defined, refers to an object that holds the substrings matching the any named capturing groups.

Unlike the String `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 either the global `g` flag or the sticky `y` flag set, it consults the `lastIndex` property of the RegExp object to determine where to start looking for a match. (And if the `y` flag is set, it also constrains the match to

begin at that position.) For a newly created RegExp object, `lastIndex` is 0, and the search begins at the start of the string. But each time `exec()` successfully finds a match, it updates the `lastIndex` property to the index of the character immediately after the matched text. If `exec()` fails to find a match, it resets `lastIndex` to 0. This special behavior allows you to call `exec()` repeatedly in order to loop through all the regular expression matches in a string. (Although, as we've described, in ES2020 and later, the `matchAll()` method of String is an easier way to loop through all matches.) For example, the loop in the following code will run twice:

```
let pattern = /Java/g;
let text = "JavaScript > Java";
let match;
while((match = pattern.exec(text)) !== null) {
    console.log(`Matched ${match[0]} at ${match.index}`);
    console.log(`Next search begins at ${pattern.lastIndex}`);
}
```

The lastIndex Property and RegExp Reuse

As you have seen already, JavaScript's regular expression API is complicated. The use of the `lastIndex` property with the `g` and `y` flags is a particularly awkward part of this API. When you use these flags, you need to be particularly careful when calling the `match()`, `exec()`, or `test()` methods because the behavior of these methods depends on `lastIndex`, and the value of `lastIndex` depends on what you have previously done with the RegExp object. This makes it easy to write buggy code.

Suppose, for example, that we wanted to find the index of all `<p>` tags within a string of HTML text. We might write code like this:

```
let match, positions = [];
while((match = /<p>/g.exec(html)) !== null) { // POSSIBLE INFINITE LOOP
    positions.push(match.index);
}
```

This code does not do what we want it to. If the `html` string contains at least one `<p>` tag, then it will loop forever. The problem is that we use a RegExp literal in the `while` loop condition. For each iteration of the loop, we're creating a new RegExp object with `lastIndex` set to 0, so `exec()` always begins at the start of the string, and if there is a match, it will keep matching over and over. The solution, of course, is to define the RegExp once, and save it to a variable so that we're using the same RegExp object for each iteration of the loop.

On the other hand, sometimes reusing a RegExp object is the wrong thing to do. Suppose, for example, that we want to loop through all of the words in a dictionary to find words that contain pairs of double letters:

```
let dictionary = [ "apple", "book", "coffee" ];
let doubleLetterWords = [];
```

```
let doubleLetter = /(\w)\1/g;

for(let word of dictionary) {
    if (doubleLetter.test(word)) {
        doubleLetterWords.push(word);
    }
}
doubleLetterWords  // => ["apple", "coffee"]: "book" is missing!
```

Because we set the `g` flag on the RegExp, the `lastIndex` property is changed after successful matches, and the `test()` method (which is based on `exec()`) starts searching for a match at the position specified by `lastIndex`. After matching the "pp" in "apple", `lastIndex` is 3, and so we start searching the word "book" at position 3 and do not see the "oo" that it contains.

We could fix this problem by removing the `g` flag (which is not actually necessary in this particular example), or by moving the RegExp literal into the body of the loop so that it is re-created on each iteration, or by explicitly resetting `lastIndex` to zero before each call to `test()`.

The moral here is that `lastIndex` makes the RegExp API error prone. So be extra careful when using the `g` or `y` flags and looping. And in ES2020 and later, use the String `matchAll()` method instead of `exec()` to sidestep this problem since `matchAll()` does not modify `lastIndex`.

11.4 Dates and Times

The Date class is JavaScript's API for working with dates and times. Create a Date object with the `Date()` constructor. With no arguments, it returns a Date object that represents the current date and time:

```
let now = new Date();      // The current time
```

If you pass one numeric argument, the `Date()` constructor interprets that argument as the number of milliseconds since the 1970 epoch:

```
let epoch = new Date(0);  // Midnight, January 1st, 1970, GMT
```

If you specify two or more integer arguments, they are interpreted as the year, month, day-of-month, hour, minute, second, and millisecond in your local time zone, as in the following:

```
let century = new Date(2100,         // Year 2100
                       0,            // January
                       1,            // 1st
                       2, 3, 4, 5);  // 02:03:04.005, local time
```

One quirk of the Date API is that the first month of a year is number 0, but the first day of a month is number 1. If you omit the time fields, the `Date()` constructor defaults them all to 0, setting the time to midnight.

Note that when invoked with multiple numbers, the `Date()` constructor interprets them using whatever time zone the local computer is set to. If you want to specify a date and time in UTC (Universal Coordinated Time, aka GMT), then you can use the `Date.UTC()`. This static method takes the same arguments as the `Date()` constructor, interprets them in UTC, and returns a millisecond timestamp that you can pass to the `Date()` constructor:

```
// Midnight in England, January 1, 2100
let century = new Date(Date.UTC(2100, 0, 1));
```

If you print a date (with `console.log(century)`, for example), it will, by default, be printed in your local time zone. If you want to display a date in UTC, you should explicitly convert it to a string with `toUTCString()` or `toISOString()`.

Finally, if you pass a string to the `Date()` constructor, it will attempt to parse that string as a date and time specification. The constructor can parse dates specified in the formats produced by the `toString()`, `toUTCString()`, and `toISOString()` methods:

```
let century = new Date("2100-01-01T00:00:00Z");  // An ISO format date
```

Once you have a Date object, various get and set methods allow you to query and modify the year, month, day-of-month, hour, minute, second, and millisecond fields of the Date. Each of these methods has two forms: one that gets or sets using local time and one that gets or sets using UTC time. To get or set the year of a Date object, for example, you would use `getFullYear()`, `getUTCFullYear()`, `setFullYear()`, or `setUTCFullYear()`:

```
let d = new Date();                 // Start with the current date
d.setFullYear(d.getFullYear() + 1); // Increment the year
```

To get or set the other fields of a Date, replace "FullYear" in the method name with "Month", "Date", "Hours", "Minutes", "Seconds", or "Milliseconds". Some of the date set methods allow you to set more than one field at a time. `setFullYear()` and `setUTCFullYear()` also optionally allow you to set the month and day-of-month as well. And `setHours()` and `setUTCHours()` allow you to specify the minutes, seconds, and milliseconds fields in addition to the hours field.

Note that the methods for querying the day-of-month are `getDate()` and `getUTCDate()`. The more natural-sounding functions `getDay()` and `getUTCDay()` return the day-of-week (0 for Sunday through 6 for Saturday). The day-of-week is read-only, so there is not a corresponding `setDay()` method.

11.4.1 Timestamps

JavaScript represents dates internally as integers that specify the number of milliseconds since (or before) midnight on January 1, 1970, UTC time. Integers as large as 8,640,000,000,000,000 are supported, so JavaScript won't be running out of milliseconds for more than 270,000 years.

For any Date object, the `getTime()` method returns this internal value, and the `setTime()` method sets it. So you can add 30 seconds to a Date with code like this, for example:

```
d.setTime(d.getTime() + 30000);
```

These millisecond values are sometimes called *timestamps*, and it is sometimes useful to work with them directly rather than with Date objects. The static `Date.now()` method returns the current time as a timestamp and is helpful when you want to measure how long your code takes to run:

```
let startTime = Date.now();
reticulateSplines(); // Do some time-consuming operation
let endTime = Date.now();
console.log(`Spline reticulation took ${endTime - startTime}ms.`);
```

High-Resolution Timestamps

The timestamps returned by `Date.now()` are measured in milliseconds. A millisecond is actually a relatively long time for a computer, and sometimes you may want to measure elapsed time with higher precision. The `performance.now()` function allows this: it also returns a millisecond-based timestamp, but the return value is not an integer, so it includes fractions of a millisecond. The value returned by `performance.now()` is not an absolute timestamp like the `Date.now()` value is. Instead, it simply indicates how much time has elapsed since a web page was loaded or since the Node process started.

The `performance` object is part of a larger Performance API that is not defined by the ECMAScript standard but is implemented by web browsers and by Node. In order to use the `performance` object in Node, you must import it with:

```
const { performance } = require("perf_hooks");
```

Allowing high-precision timing on the web may allow unscrupulous websites to fingerprint visitors, so browsers (notably Firefox) may reduce the precision of `performance.now()` by default. As a web developer, you should be able to re-enable high-precision timing somehow (such as by setting `privacy.reduceTimerPrecision` to false in Firefox).

11.4.2 Date Arithmetic

Date objects can be compared with JavaScript's standard <, <=, >, and >= comparison operators. And you can subtract one Date object from another to determine the number of milliseconds between the two dates. (This works because the Date class defines a `valueOf()` method that returns a timestamp.)

If you want to add or subtract a specified number of seconds, minutes, or hours from a Date, it is often easiest to simply modify the timestamp as demonstrated in the previous example, when we added 30 seconds to a date. This technique becomes more cumbersome if you want to add days, and it does not work at all for months and years since they have varying numbers of days. To do date arithmetic involving days, months, and years, you can use `setDate()`, `setMonth()`, and `setYear()`. Here, for example, is code that adds three months and two weeks to the current date:

```
let d = new Date();
d.setMonth(d.getMonth() + 3, d.getDate() + 14);
```

Date setting methods work correctly even when they overflow. When we add three months to the current month, we can end up with a value greater than 11 (which represents December). The `setMonth()` handles this by incrementing the year as needed. Similarly, when we set the day of the month to a value larger than the number of days in the month, the month gets incremented appropriately.

11.4.3 Formatting and Parsing Date Strings

If you are using the Date class to actually keep track of dates and times (as opposed to just measuring time intervals), then you are likely to need to display dates and times to the users of your code. The Date class defines a number of different methods for converting Date objects to strings. Here are some examples:

```
let d = new Date(2020, 0, 1, 17, 10, 30); // 5:10:30pm on New Year's Day 2020
d.toString()  // => "Wed Jan 01 2020 17:10:30 GMT-0800 (Pacific Standard Time)"
d.toUTCString()         // => "Thu, 02 Jan 2020 01:10:30 GMT"
d.toLocaleDateString()  // => "1/1/2020": 'en-US' locale
d.toLocaleTimeString()  // => "5:10:30 PM": 'en-US' locale
d.toISOString()         // => "2020-01-02T01:10:30.000Z"
```

This is a full list of the string formatting methods of the Date class:

`toString()`
: This method uses the local time zone but does not format the date and time in a locale-aware way.

`toUTCString()`
: This method uses the UTC time zone but does not format the date in a locale-aware way.

`toISOString()`
: This method prints the date and time in the standard year-month-day hours:minutes:seconds.ms format of the ISO-8601 standard. The letter "T" separates the date portion of the output from the time portion of the output. The time is expressed in UTC, and this is indicated with the letter "Z" as the last letter of the output.

`toLocaleString()`
: This method uses the local time zone and a format that is appropriate for the user's locale.

`toDateString()`
: This method formats only the date portion of the Date and omits the time. It uses the local time zone and does not do locale-appropriate formatting.

`toLocaleDateString()`
: This method formats only the date. It uses the local time zone and a locale-appropriate date format.

`toTimeString()`
: This method formats only the time and omits the date. It uses the local time zone but does not format the time in a locale-aware way.

`toLocaleTimeString()`
: This method formats the time in a locale-aware way and uses the local time zone.

None of these date-to-string methods is ideal when formatting dates and times to be displayed to end users. See §11.7.2 for a more general-purpose and locale-aware date- and time-formatting technique.

Finally, in addition to these methods that convert a Date object to a string, there is also a static `Date.parse()` method that takes a string as its argument, attempts to parse it as a date and time, and returns a timestamp representing that date. `Date.parse()` is able to parse the same strings that the `Date()` constructor can and is guaranteed to be able to parse the output of `toISOString()`, `toUTCString()`, and `toString()`.

11.5 Error Classes

The JavaScript `throw` and `catch` statements can throw and catch any JavaScript value, including primitive values. There is no exception type that must be used to signal errors. JavaScript does define an Error class, however, and it is traditional to use instances of Error or a subclass when signaling an error with `throw`. One good reason to use an Error object is that, when you create an Error, it captures the state of the JavaScript stack, and if the exception is uncaught, the stack trace will be displayed

with the error message, which will help you debug the issue. (Note that the stack trace shows where the Error object was created, not where the `throw` statement throws it. If you always create the object right before throwing it with `throw new Error()`, this will not cause any confusion.)

Error objects have two properties: `message` and `name`, and a `toString()` method. The value of the `message` property is the value you passed to the `Error()` constructor, converted to a string if necessary. For error objects created with `Error()`, the `name` property is always "Error". The `toString()` method simply returns the value of the `name` property followed by a colon and space and the value of the `message` property.

Although it is not part of the ECMAScript standard, Node and all modern browsers also define a `stack` property on Error objects. The value of this property is a multiline string that contains a stack trace of the JavaScript call stack at the moment that the Error object was created. This can be useful information to log when an unexpected error is caught.

In addition to the Error class, JavaScript defines a number of subclasses that it uses to signal particular types of errors defined by ECMAScript. These subclasses are EvalError, RangeError, ReferenceError, SyntaxError, TypeError, and URIError. You can use these error classes in your own code if they seem appropriate. Like the base Error class, each of these subclasses has a constructor that takes a single message argument. And instances of each of these subclasses have a `name` property whose value is the same as the constructor name.

You should feel free to define your own Error subclasses that best encapsulate the error conditions of your own program. Note that you are not limited to the `name` and `message` properties. If you create a subclass, you can define new properties to provide error details. If you are writing a parser, for example, you might find it useful to define a ParseError class with `line` and `column` properties that specify the exact location of the parsing failure. Or if you are working with HTTP requests, you might want to define an HTTPError class that has a `status` property that holds the HTTP status code (such as 404 or 500) of the failed request.

For example:

```
class HTTPError extends Error {
    constructor(status, statusText, url) {
        super(`${status} ${statusText}: ${url}`);
        this.status = status;
        this.statusText = statusText;
        this.url = url;
    }

    get name() { return "HTTPError"; }
}
```

```
let error = new HTTPError(404, "Not Found", "http://example.com/");
error.status        // => 404
error.message       // => "404 Not Found: http://example.com/"
error.name          // => "HTTPError"
```

11.6 JSON Serialization and Parsing

When a program needs to save data or needs to transmit data across a network connection to another program, it must to convert its in-memory data structures into a string of bytes or characters than can be saved or transmitted and then later be parsed to restore the original in-memory data structures. This process of converting data structures into streams of bytes or characters is known as *serialization* (or *marshaling* or even *pickling*).

The easiest way to serialize data in JavaScript uses a serialization format known as JSON. This acronym stands for "JavaScript Object Notation" and, as the name implies, the format uses JavaScript object and array literal syntax to convert data structures consisting of objects and arrays into strings. JSON supports primitive numbers and strings and also the values `true`, `false`, and `null`, as well as arrays and objects built up from those primitive values. JSON does not support other JavaScript types like Map, Set, RegExp, Date, or typed arrays. Nevertheless, it has proved to be a remarkably versatile data format and is in common use even with non-JavaScript-based programs.

JavaScript supports JSON serialization and deserialization with the two functions `JSON.stringify()` and `JSON.parse()`, which were covered briefly in §6.8. Given an object or array (nested arbitrarily deeply) that does not contain any nonserializable values like RegExp objects or typed arrays, you can serialize the object simply by passing it to `JSON.stringify()`. As the name implies, the return value of this function is a string. And given a string returned by `JSON.stringify()`, you can re-create the original data structure by passing the string to `JSON.parse()`:

```
let o = {s: "", n: 0, a: [true, false, null]};
let s = JSON.stringify(o);  // s == '{"s":"","n":0,"a":[true,false,null]}'
let copy = JSON.parse(s);   // copy == {s: "", n: 0, a: [true, false, null]}
```

If we leave out the part where serialized data is saved to a file or sent over the network, we can use this pair of functions as a somewhat inefficient way of creating a deep copy of an object:

```
// Make a deep copy of any serializable object or array
function deepcopy(o) {
    return JSON.parse(JSON.stringify(o));
}
```

JSON Is a Subset of JavaScript

When data is serialized to JSON format, the result is valid JavaScript source code for an expression that evaluates to a copy of the original data structure. If you prefix a JSON string with `var data =` and pass the result to `eval()`, you'll get a copy of the original data structure assigned to the variable `data`. You should never do this, however, because it is a huge security hole—if an attacker could inject arbitrary JavaScript code into a JSON file, they could make your program run their code. It is faster and safer to just use `JSON.parse()` to decode JSON-formatted data.

JSON is sometimes used as a human-readable configuration file format. If you find yourself hand-editing a JSON file, note that the JSON format is a very strict subset of JavaScript. Comments are not allowed and property names must be enclosed in double quotes even when JavaScript would not require this.

Typically, you pass only a single argument to `JSON.stringify()` and `JSON.parse()`. Both functions accept an optional second argument that allows us to extend the JSON format, and these are described next. `JSON.stringify()` also takes an optional third argument that we'll discuss first. If you would like your JSON-formatted string to be human-readable (if it is being used as a configuration file, for example), then you should pass `null` as the second argument and pass a number or string as the third argument. This third argument tells `JSON.stringify()` that it should format the data on multiple indented lines. If the third argument is a number, then it will use that number of spaces for each indentation level. If the third argument is a string of whitespace (such as `'\t'`), it will use that string for each level of indent.

```
let o = {s: "test", n: 0};
JSON.stringify(o, null, 2)  // => '{\n  "s": "test",\n  "n": 0\n}'
```

`JSON.parse()` ignores whitespace, so passing a third argument to `JSON.stringify()` has no impact on our ability to convert the string back into a data structure.

11.6.1 JSON Customizations

If `JSON.stringify()` is asked to serialize a value that is not natively supported by the JSON format, it looks to see if that value has a `toJSON()` method, and if so, it calls that method and then stringifies the return value in place of the original value. Date objects implement `toJSON()`: it returns the same string that `toISOString()` method does. This means that if you serialize an object that includes a Date, the date will automatically be converted to a string for you. When you parse the serialized string, the re-created data structure will not be exactly the same as the one you started with because it will have a string where the original object had a Date.

If you need to re-create Date objects (or modify the parsed object in any other way), you can pass a "reviver" function as the second argument to `JSON.parse()`. If specified, this "reviver" function is invoked once for each primitive value (but not the objects or arrays that contain those primitive values) parsed from the input string. The function is invoked with two arguments. The first is a property name—either an object property name or an array index converted to a string. The second argument is the primitive value of that object property or array element. Furthermore, the function is invoked as a method of the object or array that contains the primitive value, so you can refer to that containing object with the `this` keyword.

The return value of the reviver function becomes the new value of the named property. If it returns its second argument, the property will remain unchanged. If it returns `undefined`, then the named property will be deleted from the object or array before `JSON.parse()` returns to the user.

As an example, here is a call to `JSON.parse()` that uses a reviver function to filter some properties and to re-create Date objects:

```
let data = JSON.parse(text, function(key, value) {
    // Remove any values whose property name begins with an underscore
    if (key[0] === "_") return undefined;

    // If the value is a string in ISO 8601 date format convert it to a Date.
    if (typeof value === "string" &&
        /^\d\d\d\d-\d\d-\d\dT\d\d:\d\d:\d\d.\d\d\dZ$/.test(value)) {
        return new Date(value);
    }

    // Otherwise, return the value unchanged
    return value;
});
```

In addition to its use of `toJSON()` described earlier, `JSON.stringify()` also allows its output to be customized by passing an array or a function as the optional second argument.

If an array of strings (or numbers—they are converted to strings) is passed instead as the second argument, these are used as the names of object properties (or array elements). Any property whose name is not in the array will be omitted from stringification. Furthermore, the returned string will include properties in the same order that they appear in the array (which can be very useful when writing tests).

If you pass a function, it is a replacer function—effectively the inverse of the optional reviver function you can pass to `JSON.parse()`. If specified, the replacer function is invoked for each value to be stringified. The first argument to the replacer function is the object property name or array index of the value within that object, and the second argument is the value itself. The replacer function is invoked as a method of the object or array that contains the value to be stringified. The return value of the

replacer function is stringified in place of the original value. If the replacer returns `undefined` or returns nothing at all, then that value (and its array element or object property) is omitted from the stringification.

```
// Specify what fields to serialize, and what order to serialize them in
let text = JSON.stringify(address, ["city","state","country"]);

// Specify a replacer function that omits RegExp-value properties
let json = JSON.stringify(o, (k, v) => v instanceof RegExp ? undefined : v);
```

The two `JSON.stringify()` calls here use the second argument in a benign way, producing serialized output that can be deserialized without requiring a special reviver function. In general, though, if you define a `toJSON()` method for a type, or if you use a replacer function that actually replaces nonserializable values with serializable ones, then you will typically need to use a custom reviver function with `JSON.parse()` to get your original data structure back. If you do this, you should understand that you are defining a custom data format and sacrificing portability and compatibility with a large ecosystem of JSON-compatible tools and languages.

11.7 The Internationalization API

The JavaScript internationalization API consists of the three classes Intl.NumberFormat, Intl.DateTimeFormat, and Intl.Collator that allow us to format numbers (including monetary amounts and percentages), dates, and times in locale-appropriate ways and to compare strings in locale-appropriate ways. These classes are not part of the ECMAScript standard but are defined as part of the ECMA402 standard and are well-supported by web browsers. The Intl API is also supported in Node, but at the time of this writing, prebuilt Node binaries do not ship with the localization data required to make them work with locales other than US English. So in order to use these classes with Node, you may need to download a separate data package or use a custom build of Node.

One of the most important parts of internationalization is displaying text that has been translated into the user's language. There are various ways to achieve this, but none of them are within the scope of the Intl API described here.

11.7.1 Formatting Numbers

Users around the world expect numbers to be formatted in different ways. Decimal points can be periods or commas. Thousands separators can be commas or periods, and they aren't used every three digits in all places. Some currencies are divided into hundredths, some into thousandths, and some have no subdivisions. Finally, although the so-called "Arabic numerals" 0 through 9 are used in many languages, this is not universal, and users in some countries will expect to see numbers written using the digits from their own scripts.

The Intl.NumberFormat class defines a `format()` method that takes all of these formatting possibilities into account. The constructor takes two arguments. The first argument specifies the locale that the number should be formatted for and the second is an object that specifies more details about how the number should be formatted. If the first argument is omitted or `undefined`, then the system locale (which we assume to be the user's preferred locale) will be used. If the first argument is a string, it specifies a desired locale, such as `"en-US"` (English as used in the United States), `"fr"` (French), or `"zh-Hans-CN"` (Chinese, using the simplified Han writing system, in China). The first argument can also be an array of locale strings, and in this case, Intl.NumberFormat will choose the most specific one that is well supported.

The second argument to the `Intl.NumberFormat()` constructor, if specified, should be an object that defines one or more of the following properties:

`style`
: Specifies the kind of number formatting that is required. The default is `"decimal"`. Specify `"percent"` to format a number as a percentage or specify `"currency"` to specify a number as an amount of money.

`currency`
: If style is `"currency"`, then this property is required to specify the three-letter ISO currency code (such as `"USD"` for US dollars or `"GBP"` for British pounds) of the desired currency.

`currencyDisplay`
: If style is `"currency"`, then this property specifies how the currency is displayed. The default value `"symbol"` uses a currency symbol if the currency has one. The value `"code"` uses the three-letter ISO code, and the value `"name"` spells out the name of the currency in long form.

`useGrouping`
: Set this property to `false` if you do not want numbers to have thousands separators (or their locale-appropriate equivalents).

`minimumIntegerDigits`
: The minimum number of digits to use to display the integer part of the number. If the number has fewer digits than this, it will be padded on the left with zeros. The default value is 1, but you can use values as high as 21.

`minimumFractionDigits`, `maximumFractionDigits`
: These two properties control the formatting of the fractional part of the number. If a number has fewer fractional digits than the minimum, it will be padded with zeros on the right. If it has more than the maximum, then the fractional part will be rounded. Legal values for both properties are between 0 and 20. The default minimum is 0 and the default maximum is 3, except when formatting monetary

amounts, when the length of the fractional part varies depending on the specified currency.

`minimumSignificantDigits`, `maximumSignificantDigits`
: These properties control the number of significant digits used when formatting a number, making them suitable when formatting scientific data, for example. If specified, these properties override the integer and fractional digit properties listed previously. Legal values are between 1 and 21.

Once you have created an Intl.NumberFormat object with the desired locale and options, you use it by passing a number to its `format()` method, which returns an appropriately formatted string. For example:

```
let euros = Intl.NumberFormat("es", {style: "currency", currency: "EUR"});
euros.format(10)    // => "10,00 €": ten euros, Spanish formatting

let pounds = Intl.NumberFormat("en", {style: "currency", currency: "GBP"});
pounds.format(1000) // => "£1,000.00": One thousand pounds, English formatting
```

A useful feature of `Intl.NumberFormat` (and the other Intl classes as well) is that its `format()` method is bound to the NumberFormat object to which it belongs. So instead of defining a variable that refers to the formatting object and then invoking the `format()` method on that, you can just assign the `format()` method to a variable and use it as if it were a standalone function, as in this example:

```
let data = [0.05, .75, 1];
let formatData = Intl.NumberFormat(undefined, {
    style: "percent",
    minimumFractionDigits: 1,
    maximumFractionDigits: 1
}).format;

data.map(formatData)   // => ["5.0%", "75.0%", "100.0%"]: in en-US locale
```

Some languages, such as Arabic, use their own script for decimal digits:

```
let arabic = Intl.NumberFormat("ar", {useGrouping: false}).format;
arabic(1234567890)   // => "١٢٣٤٥٦٧٨٩٠"
```

Other languages, such as Hindi, use a script that has its own set of digits, but tend to use the ASCII digits 0–9 by default. If you want to override the default script used for digits, add `-u-nu-` to the locale and follow it with an abbreviated script name. You can format numbers with Indian-style grouping and Devanagari digits like this, for example:

```
let hindi = Intl.NumberFormat("hi-IN-u-nu-deva").format;
hindi(1234567890)    // => "१,२३,४५,६७,८९०"
```

`-u-` in a locale specifies that what comes next is a Unicode extension. `nu` is the extension name for the numbering system, and `deva` is short for Devanagari. The Intl API

standard defines names for a number of other numbering systems, mostly for the Indic languages of South and Southeast Asia.

11.7.2 Formatting Dates and Times

The Intl.DateTimeFormat class is a lot like the Intl.NumberFormat class. The `Intl.DateTimeFormat()` constructor takes the same two arguments that `Intl.NumberFormat()` does: a locale or array of locales and an object of formatting options. And the way you use an Intl.DateTimeFormat instance is by calling its `format()` method to convert a Date object to a string.

As mentioned in §11.4, the Date class defines simple `toLocaleDateString()` and `toLocaleTimeString()` methods that produce locale-appropriate output for the user's locale. But these methods don't give you any control over what fields of the date and time are displayed. Maybe you want to omit the year but add a weekday to the date format. Do you want the month to be represented numerically or spelled out by name? The Intl.DateTimeFormat class provides fine-grained control over what is output based on the properties in the options object that is passed as the second argument to the constructor. Note, however, that Intl.DateTimeFormat cannot always display exactly what you ask for. If you specify options to format hours and seconds but omit minutes, you'll find that the formatter displays the minutes anyway. The idea is that you use the options object to specify what date and time fields you'd like to present to the user and how you'd like those formatted (by name or by number, for example), then the formatter will look for a locale-appropriate format that most closely matches what you have asked for.

The available options are the following. Only specify properties for date and time fields that you would like to appear in the formatted output.

`year`
: Use `"numeric"` for a full, four-digit year or `"2-digit"` for a two-digit abbreviation.

`month`
: Use `"numeric"` for a possibly short number like "1", or `"2-digit"` for a numeric representation that always has two digits, like "01". Use `"long"` for a full name like "January", `"short"` for an abbreviated name like "Jan", and `"narrow"` for a highly abbreviated name like "J" that is not guaranteed to be unique.

`day`
: Use `"numeric"` for a one- or two-digit number or `"2-digit"` for a two-digit number for the day-of-month.

`weekday`
: Use `"long"` for a full name like "Monday", `"short"` for an abbreviated name like "Mon", and `"narrow"` for a highly abbreviated name like "M" that is not guaranteed to be unique.

`era`
: This property specifies whether a date should be formatted with an era, such as CE or BCE. This may be useful if you are formatting dates from very long ago or if you are using a Japanese calendar. Legal values are `"long"`, `"short"`, and `"narrow"`.

`hour`, `minute`, `second`
: These properties specify how you would like time displayed. Use `"numeric"` for a one- or two-digit field or `"2-digit"` to force single-digit numbers to be padded on the left with a 0.

`timeZone`
: This property specifies the desired time zone for which the date should be formatted. If omitted, the local time zone is used. Implementations always recognize "UTC" and may also recognize Internet Assigned Numbers Authority (IANA) time zone names, such as "America/Los_Angeles".

`timeZoneName`
: This property specifies how the time zone should be displayed in a formatted date or time. Use `"long"` for a fully spelled-out time zone name and `"short"` for an abbreviated or numeric time zone.

`hour12`
: This boolean property specifies whether or not to use 12-hour time. The default is locale dependent, but you can override it with this property.

`hourCycle`
: This property allows you to specify whether midnight is written as 0 hours, 12 hours, or 24 hours. The default is locale dependent, but you can override the default with this property. Note that `hour12` takes precedence over this property. Use the value `"h11"` to specify that midnight is 0 and the hour before midnight is 11pm. Use `"h12"` to specify that midnight is 12. Use `"h23"` to specify that midnight is 0 and the hour before midnight is 23. And use `"h24"` to specify that midnight is 24.

Here are some examples:

```
let d = new Date("2020-01-02T13:14:15Z");  // January 2nd, 2020, 13:14:15 UTC

// With no options, we get a basic numeric date format
Intl.DateTimeFormat("en-US").format(d) // => "1/2/2020"
```

```
Intl.DateTimeFormat("fr-FR").format(d) // => "02/01/2020"

// Spelled out weekday and month
let opts = { weekday: "long", month: "long", year: "numeric", day: "numeric" };
Intl.DateTimeFormat("en-US", opts).format(d) // => "Thursday, January 2, 2020"
Intl.DateTimeFormat("es-ES", opts).format(d) // => "jueves, 2 de enero de 2020"

// The time in New York, for a French-speaking Canadian
opts = { hour: "numeric", minute: "2-digit", timeZone: "America/New_York" };
Intl.DateTimeFormat("fr-CA", opts).format(d) // => "8 h 14"
```

Intl.DateTimeFormat can display dates using calendars other than the default Julian calendar based on the Christian era. Although some locales may use a non-Christian calendar by default, you can always explicitly specify the calendar to use by adding `-u-ca-` to the locale and following that with the name of the calendar. Possible calendar names include "buddhist", "chinese", "coptic", "ethiopic", "gregory", "hebrew", "indian", "islamic", "iso8601", "japanese", and "persian". Continuing the preceding example, we can determine the year in various non-Christian calendars:

```
let opts = { year: "numeric", era: "short" };
Intl.DateTimeFormat("en", opts).format(d)                // => "2020 AD"
Intl.DateTimeFormat("en-u-ca-iso8601", opts).format(d)   // => "2020 AD"
Intl.DateTimeFormat("en-u-ca-hebrew", opts).format(d)    // => "5780 AM"
Intl.DateTimeFormat("en-u-ca-buddhist", opts).format(d)  // => "2563 BE"
Intl.DateTimeFormat("en-u-ca-islamic", opts).format(d)   // => "1441 AH"
Intl.DateTimeFormat("en-u-ca-persian", opts).format(d)   // => "1398 AP"
Intl.DateTimeFormat("en-u-ca-indian", opts).format(d)    // => "1941 Saka"
Intl.DateTimeFormat("en-u-ca-chinese", opts).format(d)   // => "36 78"
Intl.DateTimeFormat("en-u-ca-japanese", opts).format(d)  // => "2 Reiwa"
```

11.7.3 Comparing Strings

The problem of sorting strings into alphabetical order (or some more general "collation order" for nonalphabetical scripts) is more challenging than English speakers often realize. English uses a relatively small alphabet with no accented letters, and we have the benefit of a character encoding (ASCII, since incorporated into Unicode) whose numerical values perfectly match our standard string sort order. Things are not so simple in other languages. Spanish, for example treats ñ as a distinct letter that comes after n and before o. Lithuanian alphabetizes Y before J, and Welsh treats digraphs like CH and DD as single letters with CH coming after C and DD sorting after D.

If you want to display strings to a user in an order that they will find natural, it is not enough use the `sort()` method on an array of strings. But if you create an Intl.Collator object, you can pass the `compare()` method of that object to the `sort()` method to perform locale-appropriate sorting of the strings. Intl.Collator objects can be configured so that the `compare()` method performs case-insensitive comparisons or even comparisons that only consider the base letter and ignore accents and other diacritics.

Like `Intl.NumberFormat()` and `Intl.DateTimeFormat()`, the `Intl.Collator()` constructor takes two arguments. The first specifies a locale or an array of locales, and the second is an optional object whose properties specify exactly what kind of string comparison is to be done. The supported properties are these:

`usage`
: This property specifies how the collator object is to be used. The default value is `"sort"`, but you can also specify `"search"`. The idea is that, when sorting strings, you typically want a collator that differentiates as many strings as possible to produce a reliable ordering. But when comparing two strings, some locales may want a less strict comparison that ignores accents, for example.

`sensitivity`
: This property specifies whether the collator is sensitive to letter case and accents when comparing strings. The value `"base"` causes comparisons that ignore case and accents, considering only the base letter for each character. (Note, however, that some languages consider certain accented characters to be distinct base letters.) `"accent"` considers accents in comparisons but ignores case. `"case"` considers case and ignores accents. And `"variant"` performs strict comparisons that consider both case and accents. The default value for this property is `"variant"` when `usage` is `"sort"`. If `usage` is `"search"`, then the default sensitivity depends on the locale.

`ignorePunctuation`
: Set this property to `true` to ignore spaces and punctuation when comparing strings. With this property set to `true`, the strings "any one" and "anyone", for example, will be considered equal.

`numeric`
: Set this property to `true` if the strings you are comparing are integers or contain integers and you want them to be sorted into numerical order instead of alphabetical order. With this option set, the string "Version 9" will be sorted before "Version 10", for example.

`caseFirst`
: This property specifies which letter case should come first. If you specify `"upper"`, then "A" will sort before "a". And if you specify `"lower"`, then "a" will sort before "A". In either case, note that the upper- and lowercase variants of the same letter will be next to one another in sort order, which is different than Unicode lexicographic ordering (the default behavior of the Array `sort()` method) in which all ASCII uppercase letters come before all ASCII lowercase letters. The default for this property is locale dependent, and implementations may ignore this property and not allow you to override the case sort order.

Once you have created an Intl.Collator object for the desired locale and options, you can use its `compare()` method to compare two strings. This method returns a number. If the returned value is less than zero, then the first string comes before the second string. If it is greater than zero, then the first string comes after the second string. And if `compare()` returns zero, then the two strings are equal as far as this collator is concerned.

This `compare()` method that takes two strings and returns a number less than, equal to, or greater than zero is exactly what the Array `sort()` method expects for its optional argument. Also, Intl.Collator automatically binds the `compare()` method to its instance, so you can pass it directly to `sort()` without having to write a wrapper function and invoke it through the collator object. Here are some examples:

```
// A basic comparator for sorting in the user's locale.
// Never sort human-readable strings without passing something like this:
const collator = new Intl.Collator().compare;
["a", "z", "A", "Z"].sort(collator)      // => ["a", "A", "z", "Z"]

// Filenames often include numbers, so we should sort those specially
const filenameOrder = new Intl.Collator(undefined, { numeric: true }).compare;
["page10", "page9"].sort(filenameOrder)  // => ["page9", "page10"]

// Find all strings that loosely match a target string
const fuzzyMatcher = new Intl.Collator(undefined, {
    sensitivity: "base",
    ignorePunctuation: true
}).compare;
let strings = ["food", "fool", "Føø Bar"];
strings.findIndex(s => fuzzyMatcher(s, "foobar") === 0)  // => 2
```

Some locales have more than one possible collation order. In Germany, for example, phone books use a slightly more phonetic sort order than dictionaries do. In Spain, before 1994, "ch" and "ll" were treated as separate letters, so that country now has a modern sort order and a traditional sort order. And in China, collation order can be based on character encodings, the base radical and strokes of each character, or on the Pinyin romanization of characters. These collation variants cannot be selected through the Intl.Collator options argument, but they can be selected by adding `-u-co-` to the locale string and adding the name of the desired variant. Use `"de-DE-u-co-phonebk"` for phone book ordering in Germany, for example, and `"zh-TW-u-co-pinyin"` for Pinyin ordering in Taiwan.

```
// Before 1994, CH and LL were treated as separate letters in Spain
const modernSpanish = Intl.Collator("es-ES").compare;
const traditionalSpanish = Intl.Collator("es-ES-u-co-trad").compare;
let palabras = ["luz", "llama", "como", "chico"];
palabras.sort(modernSpanish)      // => ["chico", "como", "llama", "luz"]
palabras.sort(traditionalSpanish) // => ["como", "chico", "luz", "llama"]
```

11.8 The Console API

You've seen the `console.log()` function used throughout this book: in web browsers, it prints a string in the "Console" tab of the browser's developer tools pane, which can be very helpful when debugging. In Node, `console.log()` is a general-purpose output function and prints its arguments to the process's stdout stream, where it typically appears to the user in a terminal window as program output.

The Console API defines a number of useful functions in addition to `console.log()`. The API is not part of any ECMAScript standard, but it is supported by browsers and by Node and has been formally written up and standardized at *https://console.spec.whatwg.org*.

The Console API defines the following functions:

`console.log()`
: This is the most well-known of the console functions. It converts its arguments to strings and outputs them to the console. It includes spaces between the arguments and starts a new line after outputting all arguments.

`console.debug()`, `console.info()`, `console.warn()`, `console.error()`
: These functions are almost identical to `console.log()`. In Node, `console.error()` sends its output to the stderr stream rather than the stdout stream, but the other functions are aliases of `console.log()`. In browsers, output messages generated by each of these functions may be prefixed by an icon that indicates its level or severity, and the developer console may also allow developers to filter console messages by level.

`console.assert()`
: If the first argument is truthy (i.e., if the assertion passes), then this function does nothing. But if the first argument is `false` or another falsy value, then the remaining arguments are printed as if they had been passed to `console.error()` with an "Assertion failed" prefix. Note that, unlike typical `assert()` functions, `console.assert()` does not throw an exception when an assertion fails.

`console.clear()`
: This function clears the console when that is possible. This works in browsers and in Node when Node is displaying its output to a terminal. If Node's output has been redirected to a file or a pipe, however, then calling this function has no effect.

`console.table()`
: This function is a remarkably powerful but little-known feature for producing tabular output, and it is particularly useful in Node programs that need to produce output that summarizes data. `console.table()` attempts to display its

argument in tabular form (although, if it can't do that, it displays it using regular `console.log()` formatting). This works best when the argument is a relatively short array of objects, and all of the objects in the array have the same (relatively small) set of properties. In this case, each object in the array is formatted as a row of the table, and each property is a column of the table. You can also pass an array of property names as an optional second argument to specify the desired set of columns. If you pass an object instead of an array of objects, then the output will be a table with one column for property names and one column for property values. Or, if those property values are themselves objects, their property names will become columns in the table.

`console.trace()`
: This function logs its arguments like `console.log()` does, and, in addition, follows its output with a stack trace. In Node, the output goes to stderr instead of stdout.

`console.count()`
: This function takes a string argument and logs that string, followed by the number of times it has been called with that string. This can be useful when debugging an event handler, for example, if you need to keep track of how many times the event handler has been triggered.

`console.countReset()`
: This function takes a string argument and resets the counter for that string.

`console.group()`
: This function prints its arguments to the console as if they had been passed to `console.log()`, then sets the internal state of the console so that all subsequent console messages (until the next `console.groupEnd()` call) will be indented relative to the message that it just printed. This allows a group of related messages to be visually grouped with indentation. In web browsers, the developer console typically allows grouped messages to be collapsed and expanded as a group. The arguments to `console.group()` are typically used to provide an explanatory name for the group.

`console.groupCollapsed()`
: This function works like `console.group()` except that in web browsers, the group will be "collapsed" by default and the messages it contains will be hidden unless the user clicks to expand the group. In Node, this function is a synonym for `console.group()`.

`console.groupEnd()`
: This function takes no arguments. It produces no output of its own but ends the indentation and grouping caused by the most recent call to `console.group()` or `console.groupCollapsed()`.

`console.time()`
: This function takes a single string argument, makes a note of the time it was called with that string, and produces no output.

`console.timeLog()`
: This function takes a string as its first argument. If that string had previously been passed to `console.time()`, then it prints that string followed by the elapsed time since the `console.time()` call. If there are any additional arguments to `console.timeLog()`, they are printed as if they had been passed to `console.log()`.

`console.timeEnd()`
: This function takes a single string argument. If that argument had previously been passed to `console.time()`, then it prints that argument and the elapsed time. After calling `console.timeEnd()`, it is no longer legal to call `console.timeLog()` without first calling `console.time()` again.

11.8.1 Formatted Output with Console

Console functions that print their arguments like `console.log()` have a little-known feature: if the first argument is a string that includes `%s`, `%i`, `%d`, `%f`, `%o`, `%O`, or `%c`, then this first argument is treated as format string,[6] and the values of subsequent arguments are substituted into the string in place of the two-character `%` sequences.

The meanings of the sequences are as follows:

`%s`
: The argument is converted to a string.

`%i` *and* `%d`
: The argument is converted to a number and then truncated to an integer.

`%f`
: The argument is converted to a number

6 C programmers will recognize many of these character sequences from the `printf()` function.

`%o` *and* `%O`
: The argument is treated as an object, and property names and values are displayed. (In web browsers, this display is typically interactive, and users can expand and collapse properties to explore a nested data structure.) `%o` and `%O` both display object details. The uppercase variant uses an implementation-dependent output format that is judged to be most useful for software developers.

`%c`
: In web browsers, the argument is interpreted as a string of CSS styles and used to style any text that follows (until the next `%c` sequence or the end of the string). In Node, the `%c` sequence and its corresponding argument are simply ignored.

Note that it is not often necessary to use a format string with the console functions: it is usually easy to obtain suitable output by simply passing one or more values (including objects) to the function and allowing the implementation to display them in a useful way. As an example, note that, if you pass an Error object to `console.log()`, it is automatically printed along with its stack trace.

11.9 URL APIs

Since JavaScript is so commonly used in web browsers and web servers, it is common for JavaScript code to need to manipulate URLs. The URL class parses URLs and also allows modification (adding search parameters or altering paths, for example) of existing URLs. It also properly handles the complicated topic of escaping and unescaping the various components of a URL.

The URL class is not part of any ECMAScript standard, but it works in Node and all internet browsers other than Internet Explorer. It is standardized at *https://url.spec.whatwg.org*.

Create a URL object with the `URL()` constructor, passing an absolute URL string as the argument. Or pass a relative URL as the first argument and the absolute URL that it is relative to as the second argument. Once you have created the URL object, its various properties allow you to query unescaped versions of the various parts of the URL:

```
let url = new URL("https://example.com:8000/path/name?q=term#fragment");
url.href        // => "https://example.com:8000/path/name?q=term#fragment"
url.origin      // => "https://example.com:8000"
url.protocol    // => "https:"
url.host        // => "example.com:8000"
url.hostname    // => "example.com"
url.port        // => "8000"
url.pathname    // => "/path/name"
url.search      // => "?q=term"
url.hash        // => "#fragment"
```

Although it is not commonly used, URLs can include a username or a username and password, and the URL class can parse these URL components, too:

```
let url = new URL("ftp://admin:1337!@ftp.example.com/");
url.href       // => "ftp://admin:1337!@ftp.example.com/"
url.origin     // => "ftp://ftp.example.com"
url.username   // => "admin"
url.password   // => "1337!"
```

The `origin` property here is a simple combination of the URL protocol and host (including the port if one is specified). As such, it is a read-only property. But each of the other properties demonstrated in the previous example is read/write: you can set any of these properties to set the corresponding part of the URL:

```
let url = new URL("https://example.com");  // Start with our server
url.pathname = "api/search";               // Add a path to an API endpoint
url.search = "q=test";                     // Add a query parameter
url.toString()  // => "https://example.com/api/search?q=test"
```

One of the important features of the URL class is that it correctly adds punctuation and escapes special characters in URLs when that is needed:

```
let url = new URL("https://example.com");
url.pathname = "path with spaces";
url.search = "q=foo#bar";
url.pathname  // => "/path%20with%20spaces"
url.search    // => "?q=foo%23bar"
url.href      // => "https://example.com/path%20with%20spaces?q=foo%23bar"
```

The `href` property in these examples is a special one: reading `href` is equivalent to calling `toString()`: it reassembles all parts of the URL into the canonical string form of the URL. And setting `href` to a new string reruns the URL parser on the new string as if you had called the `URL()` constructor again.

In the previous examples, we've been using the `search` property to refer to the entire query portion of a URL, which consists of the characters from a question mark to the end of the URL or to the first hash character. Sometimes, it is sufficient to just treat this as a single URL property. Often, however, HTTP requests encode the values of multiple form fields or multiple API parameters into the query portion of a URL using the `application/x-www-form-urlencoded` format. In this format, the query portion of the URL is a question mark followed by one or more name/value pairs, which are separated from one another by ampersands. The same name can appear more than once, resulting in a named search parameter with more than one value.

If you want to encode these kinds of name/value pairs into the query portion of a URL, then the `searchParams` property will be more useful than the `search` property. The `search` property is a read/write string that lets you get and set the entire query portion of the URL. The `searchParams` property is a read-only reference to a

URLSearchParams object, which has an API for getting, setting, adding, deleting, and sorting the parameters encoded into the query portion of the URL:

```
let url = new URL("https://example.com/search");
url.search                              // => "": no query yet
url.searchParams.append("q", "term"); // Add a search parameter
url.search                              // => "?q=term"
url.searchParams.set("q", "x");       // Change the value of this parameter
url.search                              // => "?q=x"
url.searchParams.get("q")             // => "x": query the parameter value
url.searchParams.has("q")             // => true: there is a q parameter
url.searchParams.has("p")             // => false: there is no p parameter
url.searchParams.append("opts", "1"); // Add another search parameter
url.search                              // => "?q=x&opts=1"
url.searchParams.append("opts", "&"); // Add another value for same name
url.search                              // => "?q=x&opts=1&opts=%26": note escape
url.searchParams.get("opts")          // => "1": the first value
url.searchParams.getAll("opts")       // => ["1", "&"]: all values
url.searchParams.sort();              // Put params in alphabetical order
url.search                              // => "?opts=1&opts=%26&q=x"
url.searchParams.set("opts", "y");    // Change the opts param
url.search                              // => "?opts=y&q=x"
// searchParams is iterable
[...url.searchParams]                   // => [["opts", "y"], ["q", "x"]]
url.searchParams.delete("opts");      // Delete the opts param
url.search                              // => "?q=x"
url.href                                // => "https://example.com/search?q=x"
```

The value of the `searchParams` property is a URLSearchParams object. If you want to encode URL parameters into a query string, you can create a URLSearchParams object, append parameters, then convert it to a string and set it on the `search` property of a URL:

```
let url = new URL("http://example.com");
let params = new URLSearchParams();
params.append("q", "term");
params.append("opts", "exact");
params.toString()               // => "q=term&opts=exact"
url.search = params;
url.href                        // => "http://example.com/?q=term&opts=exact"
```

11.9.1 Legacy URL Functions

Prior to the definition of the URL API described previously, there have been multiple attempts to support URL escaping and unescaping in the core JavaScript language. The first attempt was the globally defined `escape()` and `unescape()` functions, which are now deprecated but still widely implemented. They should not be used.

When `escape()` and `unescape()` were deprecated, ECMAScript introduced two pairs of alternative global functions:

`encodeURI()` *and* `decodeURI()`

`encodeURI()` takes a string as its argument and returns a new string in which non-ASCII characters plus certain ASCII characters (such as space) are escaped. `decodeURI()` reverses the process. Characters that need to be escaped are first converted to their UTF-8 encoding, then each byte of that encoding is replaced with a `%`*xx* escape sequence, where *xx* is two hexadecimal digits. Because `encodeURI()` is intended for encoding entire URLs, it does not escape URL separator characters such as `/`, `?`, and `#`. But this means that `encodeURI()` cannot work correctly for URLs that have those characters within their various components.

`encodeURIComponent()` *and* `decodeURIComponent()`

This pair of functions works just like `encodeURI()` and `decodeURI()` except that they are intended to escape individual components of a URI, so they also escape characters like `/`, `?`, and `#` that are used to separate those components. These are the most useful of the legacy URL functions, but be aware that `encodeURIComponent()` will escape `/` characters in a path name that you probably do not want escaped. And it will convert spaces in a query parameter to `%20`, even though spaces are supposed to be escaped with a `+` in that portion of a URL.

The fundamental problem with all of these legacy functions is that they seek to apply a single encoding scheme to all parts of a URL when the fact is that different portions of a URL use different encodings. If you want a properly formatted and encoded URL, the solution is simply to use the URL class for all URL manipulation you do.

11.10 Timers

Since the earliest days of JavaScript, web browsers have defined two functions—`setTimeout()` and `setInterval()`—that allow programs to ask the browser to invoke a function after a specified amount of time has elapsed or to invoke the function repeatedly at a specified interval. These functions have never been standardized as part of the core language, but they work in all browsers and in Node and are a de facto part of the JavaScript standard library.

The first argument to `setTimeout()` is a function, and the second argument is a number that specifies how many milliseconds should elapse before the function is invoked. After the specified amount of time (and maybe a little longer if the system is busy), the function will be invoked with no arguments. Here, for example, are three `setTimeout()` calls that print console messages after one second, two seconds, and three seconds:

```
setTimeout(() => { console.log("Ready..."); }, 1000);
setTimeout(() => { console.log("set..."); }, 2000);
setTimeout(() => { console.log("go!"); }, 3000);
```

Note that `setTimeout()` does not wait for the time to elapse before returning. All three lines of code in this example run almost instantly, but then nothing happens until 1,000 milliseconds elapse.

If you omit the second argument to `setTimeout()`, it defaults to 0. That does not mean, however, that the function you specify is invoked immediately. Instead, the function is registered to be called "as soon as possible." If a browser is particularly busy handling user input or other events, it may take 10 milliseconds or more before the function is invoked.

`setTimeout()` registers a function to be invoked once. Sometimes, that function will itself call `setTimeout()` to schedule another invocation at a future time. If you want to invoke a function repeatedly, however, it is often simpler to use `setInterval()`. `setInterval()` takes the same two arguments as `setTimeout()` but invokes the function repeatedly every time the specified number of milliseconds (approximately) have elapsed.

Both `setTimeout()` and `setInterval()` return a value. If you save this value in a variable, you can then use it later to cancel the execution of the function by passing it to `clearTimeout()` or `clearInterval()`. The returned value is typically a number in web browsers and is an object in Node. The actual type doesn't matter, and you should treat it as an opaque value. The only thing you can do with this value is pass it to `clearTimeout()` to cancel the execution of a function registered with `setTimeout()` (assuming it hasn't been invoked yet) or to stop the repeating execution of a function registered with `setInterval()`.

Here is an example that demonstrates the use of `setTimeout()`, `setInterval()`, and `clearInterval()` to display a simple digital clock with the Console API:

```
// Once a second: clear the console and print the current time
let clock = setInterval(() => {
    console.clear();
    console.log(new Date().toLocaleTimeString());
}, 1000);

// After 10 seconds: stop the repeating code above.
setTimeout(() => { clearInterval(clock); }, 10000);
```

We'll see `setTimeout()` and `setInterval()` again when we cover asynchronous programming in Chapter 13.

11.11 Summary

Learning a programming language is not just about mastering the grammar. It is equally important to study the standard library so that you are familiar with all the tools that are shipped with the language. This chapter has documented JavaScript's standard library, which includes:

- Important data structures, such as Set, Map, and typed arrays.
- The Date and URL classes for working with dates and URLs.
- JavaScript's regular expression grammar and its RegExp class for textual pattern matching.
- JavaScript's internationalization library for formatting dates, time, and numbers and for sorting strings.
- The `JSON` object for serializing and deserializing simple data structures and the `console` object for logging messages.

CHAPTER 12
Iterators and Generators

Iterable objects and their associated iterators are a feature of ES6 that we've seen several times throughout this book. Arrays (including TypedArrays) are iterable, as are strings and Set and Map objects. This means that the contents of these data structures can be iterated—looped over—with the `for/of` loop, as we saw in §5.4.4:

```
let sum = 0;
for(let i of [1,2,3]) { // Loop once for each of these values
    sum += i;
}
sum   // => 6
```

Iterators can also be used with the `...` operator to expand or "spread" an iterable object into an array initializer or function invocation, as we saw in §7.1.2:

```
let chars = [..."abcd"]; // chars == ["a", "b", "c", "d"]
let data = [1, 2, 3, 4, 5];
Math.max(...data)        // => 5
```

Iterators can be used with destructuring assignment:

```
let purpleHaze = Uint8Array.of(255, 0, 255, 128);
let [r, g, b, a] = purpleHaze; // a == 128
```

When you iterate a Map object, the returned values are `[key, value]` pairs, which work well with destructuring assignment in a `for/of` loop:

```
let m = new Map([["one", 1], ["two", 2]]);
for(let [k,v] of m) console.log(k, v); // Logs 'one 1' and 'two 2'
```

If you want to iterate just the keys or just the values rather than the pairs, you can use the `keys()` and `values()` methods:

```
[...m]           // => [["one", 1], ["two", 2]]: default iteration
[...m.entries()] // => [["one", 1], ["two", 2]]: entries() method is the same
```

```
[...m.keys()]     // => ["one", "two"]: keys() method iterates just map keys
[...m.values()]   // => [1, 2]: values() method iterates just map values
```

Finally, a number of built-in functions and constructors that are commonly used with Array objects are actually written (in ES6 and later) to accept arbitrary iterators instead. The `Set()` constructor is one such API:

```
// Strings are iterable, so the two sets are the same:
new Set("abc") // => new Set(["a", "b", "c"])
```

This chapter explains how iterators work and demonstrates how to create your own data structures that are iterable. After explaining basic iterators, this chapter covers generators, a powerful new feature of ES6 that is primarily used as a particularly easy way to create iterators.

12.1 How Iterators Work

The `for/of` loop and spread operator work seamlessly with iterable objects, but it is worth understanding what is actually happening to make the iteration work. There are three separate types that you need to understand to understand iteration in JavaScript. First, there are the *iterable* objects: these are types like Array, Set, and Map that can be iterated. Second, there is the *iterator* object itself, which performs the iteration. And third, there is the *iteration result* object that holds the result of each step of the iteration.

An *iterable* object is any object with a special iterator method that returns an iterator object. An *iterator* is any object with a `next()` method that returns an iteration result object. And an *iteration result* object is an object with properties named `value` and `done`. To iterate an iterable object, you first call its iterator method to get an iterator object. Then, you call the `next()` method of the iterator object repeatedly until the returned value has its `done` property set to `true`. The tricky thing about this is that the iterator method of an iterable object does not have a conventional name but uses the Symbol `Symbol.iterator` as its name. So a simple `for/of` loop over an iterable object `iterable` could also be written the hard way, like this:

```
let iterable = [99];
let iterator = iterable[Symbol.iterator]();
for(let result = iterator.next(); !result.done; result = iterator.next()) {
    console.log(result.value)  // result.value == 99
}
```

The iterator object of the built-in iterable datatypes is itself iterable. (That is, it has a method named `Symbol.iterator` that just returns itself.) This is occasionally useful in code like the following when you want to iterate though a "partially used" iterator:

```
let list = [1,2,3,4,5];
let iter = list[Symbol.iterator]();
```

```
let head = iter.next().value;  // head == 1
let tail = [...iter];          // tail == [2,3,4,5]
```

12.2 Implementing Iterable Objects

Iterable objects are so useful in ES6 that you should consider making your own data-types iterable whenever they represent something that can be iterated. The Range classes shown in Examples 9-2 and 9-3 in Chapter 9 were iterable. Those classes used generator functions to make themselves iterable. We'll document generators later in this chapter, but first, we will implement the Range class one more time, making it iterable without relying on a generator.

In order to make a class iterable, you must implement a method whose name is the Symbol `Symbol.iterator`. That method must return an iterator object that has a `next()` method. And the `next()` method must return an iteration result object that has a `value` property and/or a boolean `done` property. Example 12-1 implements an iterable Range class and demonstrates how to create iterable, iterator, and iteration result objects.

Example 12-1. An iterable numeric Range class

```
/*
 * A Range object represents a range of numbers {x: from <= x <= to}
 * Range defines a has() method for testing whether a given number is a member
 * of the range. Range is iterable and iterates all integers within the range.
 */
class Range {
    constructor (from, to) {
        this.from = from;
        this.to = to;
    }

    // Make a Range act like a Set of numbers
    has(x) { return typeof x === "number" && this.from <= x && x <= this.to; }

    // Return string representation of the range using set notation
    toString() { return `{ x | ${this.from} ≤ x ≤ ${this.to} }`; }

    // Make a Range iterable by returning an iterator object.
    // Note that the name of this method is a special symbol, not a string.
    [Symbol.iterator]() {
        // Each iterator instance must iterate the range independently of
        // others. So we need a state variable to track our location in the
        // iteration. We start at the first integer >= from.
        let next = Math.ceil(this.from);  // This is the next value we return
        let last = this.to;               // We won't return anything > this
        return {                          // This is the iterator object
            // This next() method is what makes this an iterator object.
```

```
            // It must return an iterator result object.
            next() {
                return (next <= last)   // If we haven't returned last value yet
                    ? { value: next++ } // return next value and increment it
                    : { done: true };   // otherwise indicate that we're done.
            },

            // As a convenience, we make the iterator itself iterable.
            [Symbol.iterator]() { return this; }
        };
    }
}

for(let x of new Range(1,10)) console.log(x); // Logs numbers 1 to 10
[...new Range(-2,2)]                          // => [-2, -1, 0, 1, 2]
```

In addition to making your classes iterable, it can be quite useful to define functions that return iterable values. Consider these iterable-based alternatives to the `map()` and `filter()` methods of JavaScript arrays:

```
// Return an iterable object that iterates the result of applying f()
// to each value from the source iterable
function map(iterable, f) {
    let iterator = iterable[Symbol.iterator]();
    return {     // This object is both iterator and iterable
        [Symbol.iterator]() { return this; },
        next() {
            let v = iterator.next();
            if (v.done) {
                return v;
            } else {
                return { value: f(v.value) };
            }
        }
    };
}

// Map a range of integers to their squares and convert to an array
[...map(new Range(1,4), x => x*x)]  // => [1, 4, 9, 16]

// Return an iterable object that filters the specified iterable,
// iterating only those elements for which the predicate returns true
function filter(iterable, predicate) {
    let iterator = iterable[Symbol.iterator]();
    return { // This object is both iterator and iterable
        [Symbol.iterator]() { return this; },
        next() {
            for(;;) {
                let v = iterator.next();
                if (v.done || predicate(v.value)) {
                    return v;
                }
```

```
            }
        }
    };
}

// Filter a range so we're left with only even numbers
[...filter(new Range(1,10), x => x % 2 === 0)]  // => [2,4,6,8,10]
```

One key feature of iterable objects and iterators is that they are inherently lazy: when computation is required to compute the next value, that computation can be deferred until the value is actually needed. Suppose, for example, that you have a very long string of text that you want to tokenize into space-separated words. You could simply use the `split()` method of your string, but if you do this, then the entire string has to be processed before you can use even the first word. And you end up allocating lots of memory for the returned array and all of the strings within it. Here is a function that allows you to lazily iterate the words of a string without keeping them all in memory at once (in ES2020, this function would be much easier to implement using the iterator-returning `matchAll()` method described in §11.3.2):

```
function words(s) {
    var r = /\s+|$/g;                       // Match one or more spaces or end
    r.lastIndex = s.match(/[^ ]/).index;  // Start matching at first nonspace
    return {                                // Return an iterable iterator object
        [Symbol.iterator]() {               // This makes us iterable
            return this;
        },
        next() {                            // This makes us an iterator
            let start = r.lastIndex;        // Resume where the last match ended
            if (start < s.length) {         // If we're not done
                let match = r.exec(s);      // Match the next word boundary
                if (match) {                // If we found one, return the word
                    return { value: s.substring(start, match.index) };
                }
            }
            return { done: true };          // Otherwise, say that we're done
        }
    };
}

[...words(" abc def  ghi! ")] // => ["abc", "def", "ghi!"]
```

12.2.1 "Closing" an Iterator: The Return Method

Imagine a (server-side) JavaScript variant of the `words()` iterator that, instead of taking a source string as its argument, takes the name of a file, opens the file, reads lines from it, and iterates the words from those lines. In most operating systems, programs that open files to read from them need to remember to close those files when they are done reading, so this hypothetical iterator would be sure to close the file after the `next()` method returns the last word in it.

But iterators don't always run all the way to the end: a `for/of` loop might be terminated with a `break` or `return` or by an exception. Similarly, when an iterator is used with destructuring assignment, the `next()` method is only called enough times to obtain values for each of the specified variables. The iterator may have many more values it could return, but they will never be requested.

If our hypothetical words-in-a-file iterator never runs all the way to the end, it still needs to close the file it opened. For this reason, iterator objects may implement a `return()` method to go along with the `next()` method. If iteration stops before `next()` has returned an iteration result with the `done` property set to `true` (most commonly because you left a `for/of` loop early via a `break` statement), then the interpreter will check to see if the iterator object has a `return()` method. If this method exists, the interpreter will invoke it with no arguments, giving the iterator the chance to close files, release memory, and otherwise clean up after itself. The `return()` method must return an iterator result object. The properties of the object are ignored, but it is an error to return a non-object value.

The `for/of` loop and the spread operator are really useful features of JavaScript, so when you are creating APIs, it is a good idea to use them when possible. But having to work with an iterable object, its iterator object, and the iterator's result objects makes the process somewhat complicated. Fortunately, generators can dramatically simplify the creation of custom iterators, as we'll see in the rest of this chapter.

12.3 Generators

A *generator* is a kind of iterator defined with powerful new ES6 syntax; it's particularly useful when the values to be iterated are not the elements of a data structure, but the result of a computation.

To create a generator, you must first define a *generator function*. A generator function is syntactically like a regular JavaScript function but is defined with the keyword `function*` rather than `function`. (Technically, this is not a new keyword, just a `*` after the keyword `function` and before the function name.) When you invoke a generator function, it does not actually execute the function body, but instead returns a generator object. This generator object is an iterator. Calling its `next()` method causes the body of the generator function to run from the start (or whatever its current position is) until it reaches a `yield` statement. `yield` is new in ES6 and is something like a `return` statement. The value of the `yield` statement becomes the value returned by the `next()` call on the iterator. An example makes this clearer:

```
// A generator function that yields the set of one digit (base-10) primes.
function* oneDigitPrimes() { // Invoking this function does not run the code
    yield 2;                 // but just returns a generator object. Calling
    yield 3;                 // the next() method of that generator runs
    yield 5;                 // the code until a yield statement provides
```

```
    yield 7;                // the return value for the next() method.
}

// When we invoke the generator function, we get a generator
let primes = oneDigitPrimes();

// A generator is an iterator object that iterates the yielded values
primes.next().value         // => 2
primes.next().value         // => 3
primes.next().value         // => 5
primes.next().value         // => 7
primes.next().done          // => true

// Generators have a Symbol.iterator method to make them iterable
primes[Symbol.iterator]()   // => primes

// We can use generators like other iterable types
[...oneDigitPrimes()]       // => [2,3,5,7]
let sum = 0;
for(let prime of oneDigitPrimes()) sum += prime;
sum                         // => 17
```

In this example, we used a `function*` statement to define a generator. Like regular functions, however, we can also define generators in expression form. Once again, we just put an asterisk after the `function` keyword:

```
const seq = function*(from,to) {
    for(let i = from; i <= to; i++) yield i;
};
[...seq(3,5)]  // => [3, 4, 5]
```

In classes and object literals, we can use shorthand notation to omit the `function` keyword entirely when we define methods. To define a generator in this context, we simply use an asterisk before the method name where the `function` keyword would have been, had we used it:

```
let o = {
    x: 1, y: 2, z: 3,
    // A generator that yields each of the keys of this object
    *g() {
        for(let key of Object.keys(this)) {
            yield key;
        }
    }
};
[...o.g()] // => ["x", "y", "z", "g"]
```

Note that there is no way to write a generator function using arrow function syntax.

Generators often make it particularly easy to define iterable classes. We can replace the `[Symbol.iterator]()` method show in Example 12-1 with a much shorter `*[Symbol.iterator&rbrack;()` generator function that looks like this:

```
    *[Symbol.iterator]() {
        for(let x = Math.ceil(this.from); x <= this.to; x++) yield x;
    }
```

See Example 9-3 in Chapter 9 to see this generator-based iterator function in context.

12.3.1 Generator Examples

Generators are more interesting if they actually *generate* the values they yield by doing some kind of computation. Here, for example, is a generator function that yields Fibonacci numbers:

```
function* fibonacciSequence() {
    let x = 0, y = 1;
    for(;;) {
        yield y;
        [x, y] = [y, x+y];  // Note: destructuring assignment
    }
}
```

Note that the `fibonacciSequence()` generator function here has an infinite loop and yields values forever without returning. If this generator is used with the `...` spread operator, it will loop until memory is exhausted and the program crashes. With care, it is possible to use it in a `for/of` loop, however:

```
// Return the nth Fibonacci number
function fibonacci(n) {
    for(let f of fibonacciSequence()) {
        if (n-- <= 0) return f;
    }
}
fibonacci(20)   // => 10946
```

This kind of infinite generator becomes more useful with a `take()` generator like this:

```
// Yield the first n elements of the specified iterable object
function* take(n, iterable) {
    let it = iterable[Symbol.iterator](); // Get iterator for iterable object
    while(n-- > 0) {           // Loop n times:
        let next = it.next();  // Get the next item from the iterator.
        if (next.done) return; // If there are no more values, return early
        else yield next.value; // otherwise, yield the value
    }
}

// An array of the first 5 Fibonacci numbers
[...take(5, fibonacciSequence())]  // => [1, 1, 2, 3, 5]
```

Here is another useful generator function that interleaves the elements of multiple iterable objects:

```
// Given an array of iterables, yield their elements in interleaved order.
function* zip(...iterables) {
```

```
    // Get an iterator for each iterable
    let iterators = iterables.map(i => i[Symbol.iterator]());
    let index = 0;
    while(iterators.length > 0) {       // While there are still some iterators
        if (index >= iterators.length) {    // If we reached the last iterator
            index = 0;                      // go back to the first one.
        }
        let item = iterators[index].next(); // Get next item from next iterator.
        if (item.done) {                    // If that iterator is done
            iterators.splice(index, 1);     // then remove it from the array.
        }
        else {                              // Otherwise,
            yield item.value;               // yield the iterated value
            index++;                        // and move on to the next iterator.
        }
    }
}

// Interleave three iterable objects
[...zip(oneDigitPrimes(),"ab",[0])]     // => [2,"a",0,3,"b",5,7]
```

12.3.2 yield* and Recursive Generators

In addition to the `zip()` generator defined in the preceding example, it might be useful to have a similar generator function that yields the elements of multiple iterable objects sequentially rather than interleaving them. We could write that generator like this:

```
function* sequence(...iterables) {
    for(let iterable of iterables) {
        for(let item of iterable) {
            yield item;
        }
    }
}

[...sequence("abc",oneDigitPrimes())]  // => ["a","b","c",2,3,5,7]
```

This process of yielding the elements of some other iterable object is common enough in generator functions that ES6 has special syntax for it. The `yield*` keyword is like `yield` except that, rather than yielding a single value, it iterates an iterable object and yields each of the resulting values. The `sequence()` generator function that we've used can be simplified with `yield*` like this:

```
function* sequence(...iterables) {
    for(let iterable of iterables) {
        yield* iterable;
    }
}

[...sequence("abc",oneDigitPrimes())]  // => ["a","b","c",2,3,5,7]
```

The array `forEach()` method is often an elegant way to loop over the elements of an array, so you might be tempted to write the `sequence()` function like this:

```
function* sequence(...iterables) {
    iterables.forEach(iterable => yield* iterable );  // Error
}
```

This does not work, however. `yield` and `yield*` can only be used within generator functions, but the nested arrow function in this code is a regular function, not a `function*` generator function, so `yield` is not allowed.

`yield*` can be used with any kind of iterable object, including iterables implemented with generators. This means that `yield*` allows us to define recursive generators, and you might use this feature to allow simple non-recursive iteration over a recursively defined tree structure, for example.

12.4 Advanced Generator Features

The most common use of generator functions is to create iterators, but the fundamental feature of generators is that they allow us to pause a computation, yield intermediate results, and then resume the computation later. This means that generators have features beyond those of iterators, and we explore those features in the following sections.

12.4.1 The Return Value of a Generator Function

The generator functions we've seen so far have not had `return` statements, or if they have, they have been used to cause an early return, not to return a value. Like any function, though, a generator function can return a value. In order to understand what happens in this case, recall how iteration works. The return value of the `next()` function is an object that has a `value` property and/or a `done` property. With typical iterators and generators, if the `value` property is defined, then the `done` property is undefined or is `false`. And if `done` is `true`, then `value` is undefined. But in the case of a generator that returns a value, the final call to `next` returns an object that has both `value` and `done` defined. The `value` property holds the return value of the generator function, and the `done` property is `true`, indicating that there are no more values to iterate. This final value is ignored by the `for/of` loop and by the spread operator, but it is available to code that manually iterates with explicit calls to `next()`:

```
function *oneAndDone() {
    yield 1;
    return "done";
}

// The return value does not appear in normal iteration.
[...oneAndDone()]   // => [1]
```

```
// But it is available if you explicitly call next()
let generator = oneAndDone();
generator.next()          // => { value: 1, done: false}
generator.next()          // => { value: "done", done: true }
// If the generator is already done, the return value is not returned again
generator.next()          // => { value: undefined, done: true }
```

12.4.2 The Value of a yield Expression

In the preceding discussion, we've treated `yield` as a statement that takes a value but has no value of its own. In fact, however, `yield` is an expression, and it can have a value.

When the `next()` method of a generator is invoked, the generator function runs until it reaches a `yield` expression. The expression that follows the `yield` keyword is evaluated, and that value becomes the return value of the `next()` invocation. At this point, the generator function stops executing right in the middle of evaluating the `yield` expression. The next time the `next()` method of the generator is called, the argument passed to `next()` becomes the value of the `yield` expression that was paused. So the generator returns values to its caller with `yield`, and the caller passes values in to the generator with `next()`. The generator and caller are two separate streams of execution passing values (and control) back and forth. The following code illustrates:

```
function* smallNumbers() {
    console.log("next() invoked the first time; argument discarded");
    let y1 = yield 1;    // y1 == "b"
    console.log("next() invoked a second time with argument", y1);
    let y2 = yield 2;    // y2 == "c"
    console.log("next() invoked a third time with argument", y2);
    let y3 = yield 3;    // y3 == "d"
    console.log("next() invoked a fourth time with argument", y3);
    return 4;
}

let g = smallNumbers();
console.log("generator created; no code runs yet");
let n1 = g.next("a");   // n1.value == 1
console.log("generator yielded", n1.value);
let n2 = g.next("b");   // n2.value == 2
console.log("generator yielded", n2.value);
let n3 = g.next("c");   // n3.value == 3
console.log("generator yielded", n3.value);
let n4 = g.next("d");   // n4 == { value: 4, done: true }
console.log("generator returned", n4.value);
```

When this code runs, it produces the following output that demonstrates the back-and-forth between the two blocks of code:

```
generator created; no code runs yet
next() invoked the first time; argument discarded
generator yielded 1
next() invoked a second time with argument b
generator yielded 2
next() invoked a third time with argument c
generator yielded 3
next() invoked a fourth time with argument d
generator returned 4
```

Note the asymmetry in this code. The first invocation of `next()` starts the generator, but the value passed to that invocation is not accessible to the generator.

12.4.3 The return() and throw() Methods of a Generator

We've seen that you can receive values yielded by or returned by a generator function. And you can pass values to a running generator by passing those values when you call the `next()` method of the generator.

In addition to providing input to a generator with `next()`, you can also alter the flow of control inside the generator by calling its `return()` and `throw()` methods. As the names suggest, calling these methods on a generator causes it to return a value or throw an exception as if the next statement in the generator was a `return` or `throw`.

Recall from earlier in the chapter that, if an iterator defines a `return()` method and iteration stops early, then the interpreter automatically calls the `return()` method to give the iterator a chance to close files or do other cleanup. In the case of generators, you can't define a custom `return()` method to handle cleanup, but you can structure the generator code to use a `try/finally` statement that ensures the necessary cleanup is done (in the `finally` block) when the generator returns. By forcing the generator to return, the generator's built-in `return()` method ensures that the cleanup code is run when the generator will no longer be used.

Just as the `next()` method of a generator allows us to pass arbitrary values into a running generator, the `throw()` method of a generator gives us a way to send arbitrary signals (in the form of exceptions) into a generator. Calling the `throw()` method always causes an exception inside the generator. But if the generator function is written with appropriate exception-handling code, the exception need not be fatal but can instead be a means of altering the behavior of the generator. Imagine, for example, a counter generator that yields an ever-increasing sequence of integers. This could be written so that an exception sent with `throw()` would reset the counter to zero.

When a generator uses `yield*` to yield values from some other iterable object, then a call to the `next()` method of the generator causes a call to the `next()` method of the iterable object. The same is true of the `return()` and `throw()` methods. If a generator uses `yield*` on an iterable object that has these methods defined, then calling

`return()` or `throw()` on the generator causes the iterator's `return()` or `throw()` method to be called in turn. All iterators *must* have a `next()` method. Iterators that need to clean up after incomplete iteration *should* define a `return()` method. And any iterator *may* define a `throw()` method, though I don't know of any practical reason to do so.

12.4.4 A Final Note About Generators

Generators are a very powerful generalized control structure. They give us the ability to pause a computation with `yield` and restart it again at some arbitrary later time with an arbitrary input value. It is possible to use generators to create a kind of cooperative threading system within single-threaded JavaScript code. And it is possible to use generators to mask asynchronous parts of your program so that your code appears sequential and synchronous, even though some of your function calls are actually asynchronous and depend on events from the network.

Trying to do these things with generators leads to code that is mind-bendingly hard to understand or to explain. It has been done, however, and the only really practical use case has been for managing asynchronous code. JavaScript now has `async` and `await` keywords (see Chapter 13) for this very purpose, however, and there is no longer any reason to abuse generators in this way.

12.5 Summary

In this chapter, you have learned:

- The `for/of` loop and the `...` spread operator work with iterable objects.
- An object is iterable if it has a method with the symbolic name `[Symbol.iterator]` that returns an iterator object.
- An iterator object has a `next()` method that returns an iteration result object.
- An iteration result object has a `value` property that holds the next iterated value, if there is one. If the iteration has completed, then the result object must have a `done` property set to `true`.
- You can implement your own iterable objects by defining a `[Symbol.iterator]()` method that returns an object with a `next()` method that returns iteration result objects. You can also implement functions that accept iterator arguments and return iterator values.
- Generator functions (functions defined with `function*` instead of `function`) are another way to define iterators.
- When you invoke a generator function, the body of the function does not run right away; instead, the return value is an iterable iterator object. Each time the

`next()` method of the iterator is called, another chunk of the generator function runs.

- Generator functions can use the `yield` operator to specify the values that are returned by the iterator. Each call to `next()` causes the generator function to run up to the next `yield` expression. The value of that `yield` expression then becomes the value returned by the iterator. When there are no more `yield` expressions, then the generator function returns, and the iteration is complete.

CHAPTER 13
Asynchronous JavaScript

Some computer programs, such as scientific simulations and machine learning models, are compute-bound: they run continuously, without pause, until they have computed their result. Most real-world computer programs, however, are significantly *asynchronous*. This means that they often have to stop computing while waiting for data to arrive or for some event to occur. JavaScript programs in a web browser are typically *event-driven*, meaning that they wait for the user to click or tap before they actually do anything. And JavaScript-based servers typically wait for client requests to arrive over the network before they do anything.

This kind of asynchronous programming is commonplace in JavaScript, and this chapter documents three important language features that help make it easier to work with asynchronous code. Promises, new in ES6, are objects that represent the not-yet-available result of an asynchronous operation. The keywords `async` and `await` were introduced in ES2017 and provide new syntax that simplifies asynchronous programming by allowing you to structure your Promise-based code as if it was synchronous. Finally, asynchronous iterators and the `for/await` loop were introduced in ES2018 and allow you to work with streams of asynchronous events using simple loops that appear synchronous.

Ironically, even though JavaScript provides these powerful features for working with asynchronous code, there are no features of the core language that are themselves asynchronous. In order to demonstrate Promises, `async`, `await`, and `for/await`, therefore, we will first take a detour into client-side and server-side JavaScript to explain some of the asynchronous features of web browsers and Node. (You can learn more about client-side and server-side JavaScript in Chapters 15 and 16.)

13.1 Asynchronous Programming with Callbacks

At its most fundamental level, asynchronous programming in JavaScript is done with *callbacks*. A callback is a function that you write and then pass to some other function. That other function then invokes ("calls back") your function when some condition is met or some (asynchronous) event occurs. The invocation of the callback function you provide notifies you of the condition or event, and sometimes, the invocation will include function arguments that provide additional details. This is easier to understand with some concrete examples, and the subsections that follow demonstrate various forms of callback-based asynchronous programming using both client-side JavaScript and Node.

13.1.1 Timers

One of the simplest kinds of asynchrony is when you want to run some code after a certain amount of time has elapsed. As we saw in §11.10, you can do this with the `setTimeout()` function:

```
setTimeout(checkForUpdates, 60000);
```

The first argument to `setTimeout()` is a function and the second is a time interval measured in milliseconds. In the preceding code, a hypothetical `checkForUpdates()` function will be called 60,000 milliseconds (1 minute) after the `setTimeout()` call. `checkForUpdates()` is a callback function that your program might define, and `setTimeout()` is the function that you invoke to register your callback function and specify under what asynchronous conditions it should be invoked.

`setTimeout()` calls the specified callback function one time, passing no arguments, and then forgets about it. If you are writing a function that really does check for updates, you probably want it to run repeatedly. You can do this by using `setInterval()` instead of `setTimeout()`:

```
// Call checkForUpdates in one minute and then again every minute after that
let updateIntervalId = setInterval(checkForUpdates, 60000);

// setInterval() returns a value that we can use to stop the repeated
// invocations by calling clearInterval(). (Similarly, setTimeout()
// returns a value that you can pass to clearTimeout())
function stopCheckingForUpdates() {
    clearInterval(updateIntervalId);
}
```

13.1.2 Events

Client-side JavaScript programs are almost universally event driven: rather than running some kind of predetermined computation, they typically wait for the user to do something and then respond to the user's actions. The web browser generates an *event* when the user presses a key on the keyboard, moves the mouse, clicks a mouse button, or touches a touchscreen device. Event-driven JavaScript programs register callback functions for specified types of events in specified contexts, and the web browser invokes those functions whenever the specified events occur. These callback functions are called *event handlers* or *event listeners*, and they are registered with `addEventListener()`:

```
// Ask the web browser to return an object representing the HTML
// <button> element that matches this CSS selector
let okay = document.querySelector('#confirmUpdateDialog button.okay');

// Now register a callback function to be invoked when the user
// clicks on that button.
okay.addEventListener('click', applyUpdate);
```

In this example, `applyUpdate()` is a hypothetical callback function that we assume is implemented somewhere else. The call to `document.querySelector()` returns an object that represents a single specified element in the web page. We call `addEventListener()` on that element to register our callback. Then the first argument to `addEventListener()` is a string that specifies the kind of event we're interested in—a mouse click or touchscreen tap, in this case. If the user clicks or taps on that specific element of the web page, then the browser will invoke our `applyUpdate()` callback function, passing an object that includes details (such as the time and the mouse pointer coordinates) about the event.

13.1.3 Network Events

Another common source of asynchrony in JavaScript programming is network requests. JavaScript running in the browser can fetch data from a web server with code like this:

```
function getCurrentVersionNumber(versionCallback) { // Note callback argument
    // Make a scripted HTTP request to a backend version API
    let request = new XMLHttpRequest();
    request.open("GET", "http://www.example.com/api/version");
    request.send();

    // Register a callback that will be invoked when the response arrives
    request.onload = function() {
        if (request.status === 200) {
            // If HTTP status is good, get version number and call callback.
            let currentVersion = parseFloat(request.responseText);
            versionCallback(null, currentVersion);
```

```
            } else {
                // Otherwise report an error to the callback
                versionCallback(response.statusText, null);
            }
        };
        // Register another callback that will be invoked for network errors
        request.onerror = request.ontimeout = function(e) {
            versionCallback(e.type, null);
        };
    }
```

Client-side JavaScript code can use the XMLHttpRequest class plus callback functions to make HTTP requests and asynchronously handle the server's response when it arrives.[1] The `getCurrentVersionNumber()` function defined here (we can imagine that it is used by the hypothetical `checkForUpdates()` function we discussed in §13.1.1) makes an HTTP request and defines event handlers that will be invoked when the server's response is received or when a timeout or other error causes the request to fail.

Notice that the code example above does not call `addEventListener()` as our previous example did. For most web APIs (including this one), event handlers can be defined by invoking `addEventListener()` on the object generating the event and passing the name of the event of interest along with the callback function. Typically, though, you can also register a single event listener by assigning it directly to a property of the object. That is what we do in this example code, assigning functions to the `onload`, `onerror`, and `ontimeout` properties. By convention, event listener properties like these always have names that begin with *on*. `addEventListener()` is the more flexible technique because it allows for multiple event handlers. But in cases where you are sure that no other code will need to register a listener for the same object and event type, it can be simpler to simply set the appropriate property to your callback.

Another thing to note about the `getCurrentVersionNumber()` function in this example code is that, because it makes an asynchronous request, it cannot synchronously return the value (the current version number) that the caller is interested in. Instead, the caller passes a callback function, which is invoked when the result is ready or when an error occurs. In this case, the caller supplies a callback function that expects two arguments. If the XMLHttpRequest works correctly, then `getCurrentVersionNumber()` invokes the callback with a `null` first argument and the version number as the second argument. Or, if an error occurs, then `getCurrentVersionNumber()` invokes the callback with error details in the first argument and `null` as the second argument.

1 The XMLHttpRequest class has nothing in particular to do with XML. In modern client-side JavaScript, it has largely been replaced by the `fetch()` API, which is covered in §15.11.1. The code example shown here is the last XMLHttpRequest-based example remaining in this book.

13.1.4 Callbacks and Events in Node

The Node.js server-side JavaScript environment is deeply asynchronous and defines many APIs that use callbacks and events. The default API for reading the contents of a file, for example, is asynchronous and invokes a callback function when the contents of the file have been read:

```
const fs = require("fs"); // The "fs" module has filesystem-related APIs
let options = {           // An object to hold options for our program
    // default options would go here
};

// Read a configuration file, then call the callback function
fs.readFile("config.json", "utf-8", (err, text) => {
    if (err) {
        // If there was an error, display a warning, but continue
        console.warn("Could not read config file:", err);
    } else {
        // Otherwise, parse the file contents and assign to the options object
        Object.assign(options, JSON.parse(text));
    }

    // In either case, we can now start running the program
    startProgram(options);
});
```

Node's `fs.readFile()` function takes a two-parameter callback as its last argument. It reads the specified file asynchronously and then invokes the callback. If the file was read successfully, it passes the file contents as the second callback argument. If there was an error, it passes the error as the first callback argument. In this example, we express the callback as an arrow function, which is a succinct and natural syntax for this kind of simple operation.

Node also defines a number of event-based APIs. The following function shows how to make an HTTP request for the contents of a URL in Node. It has two layers of asynchronous code handled with event listeners. Notice that Node uses an `on()` method to register event listeners instead of `addEventListener()`:

```
const https = require("https");

// Read the text content of the URL and asynchronously pass it to the callback.
function getText(url, callback) {
    // Start an HTTP GET request for the URL
    request = https.get(url);

    // Register a function to handle the "response" event.
    request.on("response", response => {
        // The response event means that response headers have been received
        let httpStatus = response.statusCode;
```

```
            // The body of the HTTP response has not been received yet.
            // So we register more event handlers to to be called when it arrives.
            response.setEncoding("utf-8");  // We're expecting Unicode text
            let body = "";                  // which we will accumulate here.

            // This event handler is called when a chunk of the body is ready
            response.on("data", chunk => { body += chunk; });

            // This event handler is called when the response is complete
            response.on("end", () => {
                if (httpStatus === 200) {   // If the HTTP response was good
                    callback(null, body);   // Pass response body to the callback
                } else {                    // Otherwise pass an error
                    callback(httpStatus, null);
                }
            });
        });

        // We also register an event handler for lower-level network errors
        request.on("error", (err) => {
            callback(err, null);
        });
    }
```

13.2 Promises

Now that we've seen examples of callback and event-based asynchronous programming in client-side and server-side JavaScript environments, we can introduce *Promises*, a core language feature designed to simplify asynchronous programming.

A Promise is an object that represents the result of an asynchronous computation. That result may or may not be ready yet, and the Promise API is intentionally vague about this: there is no way to synchronously get the value of a Promise; you can only ask the Promise to call a callback function when the value is ready. If you are defining an asynchronous API like the `getText()` function in the previous section, but want to make it Promise-based, omit the callback argument, and instead return a Promise object. The caller can then register one or more callbacks on this Promise object, and they will be invoked when the asynchronous computation is done.

So, at the simplest level, Promises are just a different way of working with callbacks. However, there are practical benefits to using them. One real problem with callback-based asynchronous programming is that it is common to end up with callbacks inside callbacks inside callbacks, with lines of code so highly indented that it is difficult to read. Promises allow this kind of nested callback to be re-expressed as a more linear *Promise chain* that tends to be easier to read and easier to reason about.

Another problem with callbacks is that they can make handling errors difficult. If an asynchronous function (or an asynchronously invoked callback) throws an exception,

there is no way for that exception to propagate back to the initiator of the asynchronous operation. This is a fundamental fact about asynchronous programming: it breaks exception handling. The alternative is to meticulously track and propagate errors with callback arguments and return values, but this is tedious and difficult to get right. Promises help here by standardizing a way to handle errors and providing a way for errors to propagate correctly through a chain of promises.

Note that Promises represent the future results of single asynchronous computations. They cannot be used to represent repeated asynchronous computations, however. Later in this chapter, we'll write a Promise-based alternative to the `setTimeout()` function, for example. But we can't use Promises to replace `setInterval()` because that function invokes a callback function repeatedly, which is something that Promises are just not designed to do. Similarly, we could use a Promise instead of the "load" event handler of an XMLHttpRequest object, since that callback is only ever called once. But we typically would not use a Promise in place of a "click" event handler of an HTML button object, since we normally want to allow the user to click a button multiple times.

The subsections that follow will:

- Explain Promise terminology and show basic Promise usage
- Show how promises can be chained
- Demonstrate how to create your own Promise-based APIs

Promises seem simple at first, and the basic use case for Promises is, in fact, straightforward and simple. But they can become surprisingly confusing for anything beyond the simplest use cases. Promises are a powerful idiom for asynchronous programming, but you need to understand them deeply to use them correctly and confidently. It is worth taking the time to develop that deep understanding, however, and I urge you to study this long chapter carefully.

13.2.1 Using Promises

With the advent of Promises in the core JavaScript language, web browsers have begun to implement Promise-based APIs. In the previous section, we implemented a `getText()` function that made an asynchronous HTTP request and passed the body of the HTTP response to a specified callback function as a string. Imagine a variant of this function, `getJSON()`, which parses the body of the HTTP response as JSON and returns a Promise instead of accepting a callback argument. We will implement a

`getJSON()` function later in this chapter, but for now, let's look at how we would use this Promise-returning utility function:

```
getJSON(url).then(jsonData => {
    // This is a callback function that will be asynchronously
    // invoked with the parsed JSON value when it becomes available.
});
```

`getJSON()` starts an asynchronous HTTP request for the URL you specify and then, while that request is pending, it returns a Promise object. The Promise object defines a `then()` instance method. Instead of passing our callback function directly to `getJSON()`, we instead pass it to the `then()` method. When the HTTP response arrives, the body of that response is parsed as JSON, and the resulting parsed value is passed to the function that we passed to `then()`.

You can think of the `then()` method as a callback registration method like the `addEventListener()` method used for registering event handlers in client-side JavaScript. If you call the `then()` method of a Promise object multiple times, each of the functions you specify will be called when the promised computation is complete.

Unlike many event listeners, though, a Promise represents a single computation, and each function registered with `then()` will be invoked only once. It is worth noting that the function you pass to `then()` is invoked asynchronously, even if the asynchronous computation is already complete when you call `then()`.

At a simple syntactical level, the `then()` method is the distinctive feature of Promises, and it is idiomatic to append `.then()` directly to the function invocation that returns the Promise, without the intermediate step of assigning the Promise object to a variable.

It is also idiomatic to name functions that return Promises and functions that use the results of Promises with verbs, and these idioms lead to code that is particularly easy to read:

```
// Suppose you have a function like this to display a user profile
function displayUserProfile(profile) { /* implementation omitted */ }

// Here's how you might use that function with a Promise.
// Notice how this line of code reads almost like an English sentence:
getJSON("/api/user/profile").then(displayUserProfile);
```

Handling errors with Promises

Asynchronous operations, particularly those that involve networking, can typically fail in a number of ways, and robust code has to be written to handle the errors that will inevitably occur.

For Promises, we can do this by passing a second function to the `then()` method:

```
getJSON("/api/user/profile").then(displayUserProfile, handleProfileError);
```

A Promise represents the future result of an asynchronous computation that occurs after the Promise object is created. Because the computation is performed after the Promise object is returned to us, there is no way that the computation can traditionally return a value or throw an exception that we can catch. The functions that we pass to `then()` provide alternatives. When a synchronous computation completes normally, it simply returns its result to its caller. When a Promise-based asynchronous computation completes normally, it passes its result to the function that is the first argument to `then()`.

When something goes wrong in a synchronous computation, it throws an exception that propagates up the call stack until there is a `catch` clause to handle it. When an asynchronous computation runs, its caller is no longer on the stack, so if something goes wrong, it is simply not possible to throw an exception back to the caller.

Instead, Promise-based asynchronous computations pass the exception (typically as an Error object of some kind, though this is not required) to the second function passed to `then()`. So, in the code above, if `getJSON()` runs normally, it passes its result to `displayUserProfile()`. If there is an error (the user is not logged in, the server is down, the user's internet connection dropped, the request timed out, etc.), then `getJSON()` passes an Error object to `handleProfileError()`.

In practice, it is rare to see two functions passed to `then()`. There is a better and more idiomatic way of handling errors when working with Promises. To understand it, first consider what happens if `getJSON()` completes normally but an error occurs in `displayUserProfile()`. That callback function is invoked asynchronously when `getJSON()` returns, so it is also asynchronous and cannot meaningfully throw an exception (because there is no code on the call stack to handle it).

The more idiomatic way to handle errors in this code looks like this:

```
getJSON("/api/user/profile").then(displayUserProfile).catch(handleProfileError);
```

With this code, a normal result from `getJSON()` is still passed to `displayUserProfile()`, but any error in `getJSON()` or in `displayUserProfile()` (including any exceptions thrown by `displayUserProfile`) get passed to `handleProfileError()`. The `catch()` method is just a shorthand for calling `then()` with a `null` first argument and the specified error handler function as the second argument.

We'll have more to say about `catch()` and this error-handling idiom when we discuss Promise chains in the next section.

Promise Terminology

Before we discuss Promises further, it is worth pausing to define some terms. When we are not programming and we talk about human promises, we say that a promise is "kept" or "broken." When discussing JavaScript Promises, the equivalent terms are "fulfilled" and "rejected." Imagine that you have called the `then()` method of a Promise and have passed two callback functions to it. We say that the promise has been *fulfilled* if and when the first callback is called. And we say that the Promise has been *rejected* if and when the second callback is called. If a Promise is neither fulfilled nor rejected, then it is *pending*. And once a promise is fulfilled or rejected, we say that it is *settled*. Note that a Promise can never be both fulfilled *and* rejected. Once a Promise settles, it will never change from fulfilled to rejected or vice versa.

Remember how we defined Promises at the start of this section: "a Promise is an object that represents the *result* of an asynchronous operation." It is important to remember that Promises are not just abstract ways registering callbacks to run when some async code finishes—they represent the results of that async code. If the async code runs normally (and the Promise is fulfilled), then that result is essentially the return value of the code. And if the async code does not complete normally (and the Promise is rejected), then the result is an Error object or some other value that the code might have thrown if it was not asynchronous. Any Promise that has settled has a value associated with it, and that value will not change. If the Promise is fulfilled, then the value is a return value that gets passed to any callback functions registered as the first argument of `then()`. If the Promise is rejected, then the value is an error of some sort that is passed to any callback functions registered with `catch()` or as the second argument of `then()`.

The reason that I want to be precise about Promise terminology is that Promises can also be *resolved*. It is easy to confuse this resolved state with the fulfilled state or with settled state, but it is not precisely the same as either. Understanding the resolved state is one of the keys to a deep understanding of Promises, and I'll come back to it after we've discussed Promise chains below.

13.2.2 Chaining Promises

One of the most important benefits of Promises is that they provide a natural way to express a sequence of asynchronous operations as a linear chain of `then()` method invocations, without having to nest each operation within the callback of the previous one. Here, for example, is a hypothetical Promise chain:

```
fetch(documentURL)                      // Make an HTTP request
    .then(response => response.json())  // Ask for the JSON body of the response
    .then(document => {                 // When we get the parsed JSON
        return render(document);        // display the document to the user
    })
```

```
    .then(rendered => {                 // When we get the rendered document
        cacheInDatabase(rendered);      // cache it in the local database.
    })
    .catch(error => handle(error));     // Handle any errors that occur
```

This code illustrates how a chain of Promises can make it easy to express a sequence of asynchronous operations. We're not going to discuss this particular Promise chain at all, however. We will continue to explore the idea of using Promise chains to make HTTP requests, however.

Earlier in this chapter, we saw the XMLHttpRequest object used to make an HTTP request in JavaScript. That strangely named object has an old and awkward API, and it has largely been replaced by the newer, Promise-based Fetch API (§15.11.1). In its simplest form, this new HTTP API is just the function `fetch()`. You pass it a URL, and it returns a Promise. That promise is fulfilled when the HTTP response begins to arrive and the HTTP status and headers are available:

```
fetch("/api/user/profile").then(response => {
    // When the promise resolves, we have status and headers
    if (response.ok &&
        response.headers.get("Content-Type") === "application/json") {
        // What can we do here? We don't actually have the response body yet.
    }
});
```

When the Promise returned by `fetch()` is fulfilled, it passes a Response object to the function you passed to its `then()` method. This response object gives you access to request status and headers, and it also defines methods like `text()` and `json()`, which give you access to the body of the response in text and JSON-parsed forms, respectively. But although the initial Promise is fulfilled, the body of the response may not yet have arrived. So these `text()` and `json()` methods for accessing the body of the response themselves return Promises. Here's a naive way of using `fetch()` and the `response.json()` method to get the body of an HTTP response:

```
fetch("/api/user/profile").then(response => {
    response.json().then(profile => {  // Ask for the JSON-parsed body
        // When the body of the response arrives, it will be automatically
        // parsed as JSON and passed to this function.
        displayUserProfile(profile);
    });
});
```

This is a naive way to use Promises because we nested them, like callbacks, which defeats the purpose. The preferred idiom is to use Promises in a sequential chain with code like this:

```
fetch("/api/user/profile")
    .then(response => {
        return response.json();
    })
```

```
    .then(profile => {
        displayUserProfile(profile);
    });
```

Let's look at the method invocations in this code, ignoring the arguments that are passed to the methods:

```
fetch().then().then()
```

When more than one method is invoked in a single expression like this, we call it a *method chain*. We know that the `fetch()` function returns a Promise object, and we can see that the first `.then()` in this chain invokes a method on that returned Promise object. But there is a second `.then()` in the chain, which means that the first invocation of the `then()` method must itself return a Promise.

Sometimes, when an API is designed to use this kind of method chaining, there is just a single object, and each method of that object returns the object itself in order to facilitate chaining. That is not how Promises work, however. When we write a chain of `.then()` invocations, we are not registering multiple callbacks on a single Promise object. Instead, each invocation of the `then()` method returns a new Promise object. That new Promise object is not fulfilled until the function passed to `then()` is complete.

Let's return to a simplified form of the original `fetch()` chain above. If we define the functions passed to the `then()` invocations elsewhere, we might refactor the code to look like this:

```
fetch(theURL)          // task 1; returns promise 1
    .then(callback1)   // task 2; returns promise 2
    .then(callback2);  // task 3; returns promise 3
```

Let's walk through this code in detail:

1. On the first line, `fetch()` is invoked with a URL. It initiates an HTTP GET request for that URL and returns a Promise. We'll call this HTTP request "task 1" and we'll call the Promise "promise 1".
2. On the second line, we invoke the `then()` method of promise 1, passing the `callback1` function that we want to be invoked when promise 1 is fulfilled. The `then()` method stores our callback somewhere, then returns a new Promise. We'll call the new Promise returned at this step "promise 2", and we'll say that "task 2" begins when `callback1` is invoked.
3. On the third line, we invoke the `then()` method of promise 2, passing the `callback2` function we want invoked when promise 2 is fulfilled. This `then()` method remembers our callback and returns yet another Promise. We'll say that "task 3" begins when `callback2` is invoked. We can call this latest Promise "promise 3", but we don't really need a name for it because we won't be using it at all.

4. The previous three steps all happen synchronously when the expression is first executed. Now we have an asynchronous pause while the HTTP request initiated in step 1 is sent out across the internet.
5. Eventually, the HTTP response starts to arrive. The asynchronous part of the `fetch()` call wraps the HTTP status and headers in a Response object and fulfills promise 1 with that Response object as the value.
6. When promise 1 is fulfilled, its value (the Response object) is passed to our `callback1()` function, and task 2 begins. The job of this task, given a Response object as input, is to obtain the response body as a JSON object.
7. Let's assume that task 2 completes normally and is able to parse the body of the HTTP response to produce a JSON object. This JSON object is used to fulfill promise 2.
8. The value that fulfills promise 2 becomes the input to task 3 when it is passed to the `callback2()` function. This third task now displays the data to the user in some unspecified way. When task 3 is complete (assuming it completes normally), then promise 3 will be fulfilled. But because we never did anything with promise 3, nothing happens when that Promise settles, and the chain of asynchronous computation ends at this point.

13.2.3 Resolving Promises

While explaining the URL-fetching Promise chain with the list in the last section, we talked about promises 1, 2, and 3. But there is actually a fourth Promise object involved as well, and this brings us to our important discussion of what it means for a Promise to be "resolved."

Remember that `fetch()` returns a Promise object which, when fulfilled, passes a Response object to the callback function we register. This Response object has `.text()`, `.json()`, and other methods to request the body of the HTTP response in various forms. But since the body may not yet have arrived, these methods must return Promise objects. In the example we've been studying, "task 2" calls the `.json()` method and returns its value. This is the fourth Promise object, and it is the return value of the `callback1()` function.

Let's rewrite the URL-fetching code one more time in a verbose and nonidiomatic way that makes the callbacks and promises explicit:

```
function c1(response) {              // callback 1
    let p4 = response.json();
    return p4;                       // returns promise 4
}

function c2(profile) {               // callback 2
```

```
    displayUserProfile(profile);
}

let p1 = fetch("/api/user/profile");  // promise 1, task 1
let p2 = p1.then(c1);                 // promise 2, task 2
let p3 = p2.then(c2);                 // promise 3, task 3
```

In order for Promise chains to work usefully, the output of task 2 must become the input to task 3. And in the example we're considering here, the input to task 3 is the body of the URL that was fetched, parsed as a JSON object. But, as we've just discussed, the return value of callback `c1` is not a JSON object, but Promise `p4` for that JSON object. This seems like a contradiction, but it is not: when `p1` is fulfilled, `c1` is invoked, and task 2 begins. And when `p2` is fulfilled, `c2` is invoked, and task 3 begins. But just because task 2 begins when `c1` is invoked, it does not mean that task 2 must end when `c1` returns. Promises are about managing asynchronous tasks, after all, and if task 2 is asynchronous (which it is, in this case), then that task will not be complete by the time the callback returns.

We are now ready to discuss the final detail that you need to understand to really master Promises. When you pass a callback `c` to the `then()` method, `then()` returns a Promise `p` and arranges to asynchronously invoke `c` at some later time. The callback performs some computation and returns a value `v`. When the callback returns, `p` is *resolved* with the value `v`. When a Promise is resolved with a value that is not itself a Promise, it is immediately fulfilled with that value. So if `c` returns a non-Promise, that return value becomes the value of `p`, `p` is fulfilled and we are done. But if the return value `v` is itself a Promise, then `p` is *resolved but not yet fulfilled*. At this stage, `p` cannot settle until the Promise `v` settles. If `v` is fulfilled, then `p` will be fulfilled to the same value. If `v` is rejected, then `p` will be rejected for the same reason. This is what the "resolved" state of a Promise means: the Promise has become associated with, or "locked onto," another Promise. We don't know yet whether `p` will be fulfilled or rejected, but our callback `c` no longer has any control over that. `p` is "resolved" in the sense that its fate now depends entirely on what happens to Promise `v`.

Let's bring this back to our URL-fetching example. When `c1` returns `p4`, `p2` is resolved. But being resolved is not the same as being fulfilled, so task 3 does not begin yet. When the full body of the HTTP response becomes available, then the `.json()` method can parse it and use that parsed value to fulfill `p4`. When `p4` is fulfilled, `p2` is automatically fulfilled as well, with the same parsed JSON value. At this point, the parsed JSON object is passed to `c2`, and task 3 begins.

This can be one of the trickiest parts of JavaScript to understand, and you may need to read this section more than once. Figure 13-1 presents the process in visual form and may help clarify it for you.

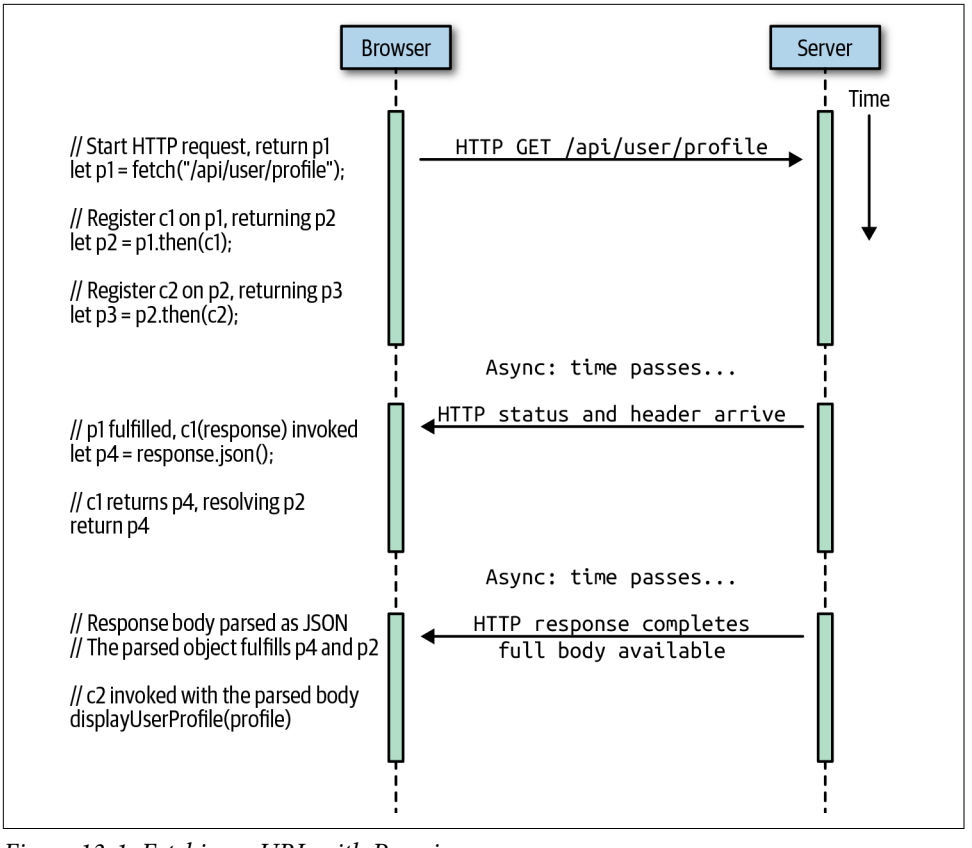

Figure 13-1. Fetching a URL with Promises

13.2.4 More on Promises and Errors

Earlier in the chapter, we saw that you can pass a second callback function to the `.then()` method and that this second function will be invoked if the Promise is rejected. When that happens, the argument to this second callback function is a value—typically an Error object—that represents the reason for the rejection. We also learned that it is uncommon (and even unidiomatic) to pass two callbacks to a `.then()` method. Instead, Promise-related errors are typically handled by adding a `.catch()` method invocation to a Promise chain. Now that we have examined Promise chains, we can return to error handling and discuss it in more detail. To preface the discussion, I'd like to stress that careful error handling is really important when doing asynchronous programming. With synchronous code, if you leave out error-handling code, you'll at least get an exception and a stack trace that you can use to figure out what is going wrong. With asynchronous code, unhandled exceptions will often go unreported, and errors can occur silently, making them much harder to

debug. The good news is that the `.catch()` method makes it easy to handle errors when working with Promises.

The catch and finally methods

The `.catch()` method of a Promise is simply a shorthand way to call `.then()` with `null` as the first argument and an error-handling callback as the second argument. Given any Promise `p` and a callback `c`, the following two lines are equivalent:

```
p.then(null, c);
p.catch(c);
```

The `.catch()` shorthand is preferred because it is simpler and because the name matches the `catch` clause in a `try/catch` exception-handling statement. As we've discussed, normal exceptions don't work with asynchronous code. The `.catch()` method of Promises is an alternative that does work for asynchronous code. When something goes wrong in synchronous code, we can speak of an exception "bubbling up the call stack" until it finds a `catch` block. With an asynchronous chain of Promises, the comparable metaphor might be of an error "trickling down the chain" until it finds a `.catch()` invocation.

In ES2018, Promise objects also define a `.finally()` method whose purpose is similar to the `finally` clause in a `try/catch/finally` statement. If you add a `.finally()` invocation to your Promise chain, then the callback you pass to `.finally()` will be invoked when the Promise you called it on settles. Your callback will be invoked if the Promise fulfills or rejects, and it will not be passed any arguments, so you can't find out whether it fulfilled or rejected. But if you need to run some kind of cleanup code (such as closing open files or network connections) in either case, a `.finally()` callback is the ideal way to do that. Like `.then()` and `.catch()`, `.finally()` returns a new Promise object. The return value of a `.finally()` callback is generally ignored, and the Promise returned by `.finally()` will typically resolve or reject with the same value that the Promise that `.finally()` was invoked on resolves or rejects with. If a `.finally()` callback throws an exception, however, then the Promise returned by `.finally()` will reject with that value.

The URL-fetching code that we studied in the previous sections did not do any error handling. Let's correct that now with a more realistic version of the code:

```
fetch("/api/user/profile")    // Start the HTTP request
    .then(response => {       // Call this when status and headers are ready
        if (!response.ok) {   // If we got a 404 Not Found or similar error
            return null;      // Maybe user is logged out; return null profile
        }

        // Now check the headers to ensure that the server sent us JSON.
        // If not, our server is broken, and this is a serious error!
        let type = response.headers.get("content-type");
```

```
        if (type !== "application/json") {
            throw new TypeError(`Expected JSON, got ${type}`);
        }

        // If we get here, then we got a 2xx status and a JSON content-type
        // so we can confidently return a Promise for the response
        // body as a JSON object.
        return response.json();
    })
    .then(profile => {        // Called with the parsed response body or null
        if (profile) {
            displayUserProfile(profile);
        }
        else { // If we got a 404 error above and returned null we end up here
            displayLoggedOutProfilePage();
        }
    })
    .catch(e => {
        if (e instanceof NetworkError) {
            // fetch() can fail this way if the internet connection is down
            displayErrorMessage("Check your internet connection.");
        }
        else if (e instanceof TypeError) {
            // This happens if we throw TypeError above
            displayErrorMessage("Something is wrong with our server!");
        }
        else {
            // This must be some kind of unanticipated error
            console.error(e);
        }
    });
```

Let's analyze this code by looking at what happens when things go wrong. We'll use the naming scheme we used before: `p1` is the Promise returned by the `fetch()` call. `p2` is the Promise returned by the first `.then()` call, and `c1` is the callback that we pass to that `.then()` call. `p3` is the Promise returned by the second `.then()` call, and `c2` is the callback we pass to that call. Finally, `c3` is the callback that we pass to the `.catch()` call. (That call returns a Promise, but we don't need to refer to it by name.)

The first thing that could fail is the `fetch()` request itself. If the network connection is down (or for some other reason an HTTP request cannot be made), then Promise `p1` will be rejected with a NetworkError object. We didn't pass an error-handling callback function as the second argument to the `.then()` call, so `p2` rejects as well with the same NetworkError object. (If we had passed an error handler to that first `.then()` call, the error handler would be invoked, and if it returned normally, `p2` would be resolved and/or fulfilled with the return value from that handler.) Without a handler, though, `p2` is rejected, and then `p3` is rejected for the same reason. At this point, the `c3` error-handling callback is called, and the NetworkError-specific code within it runs.

Another way our code could fail is if our HTTP request returns a 404 Not Found or another HTTP error. These are valid HTTP responses, so the `fetch()` call does not consider them errors. `fetch()` encapsulates a 404 Not Found in a Response object and fulfills `p1` with that object, causing `c1` to be invoked. Our code in `c1` checks the `ok` property of the Response object to detect that it has not received a normal HTTP response and handles that case by simply returning `null`. Because this return value is not a Promise, it fulfills `p2` right away, and `c2` is invoked with this value. Our code in `c2` explicitly checks for and handles falsy values by displaying a different result to the user. This is a case where we treat an abnormal condition as a nonerror and handle it without actually using an error handler.

A more serious error occurs in `c1` if the we get a normal HTTP response code but the Content-Type header is not set appropriately. Our code expects a JSON-formatted response, so if the server is sending us HTML, XML, or plain text instead, we're going to have a problem. `c1` includes code to check the Content-Type header. If the header is wrong, it treats this as a nonrecoverable problem and throws a TypeError. When a callback passed to `.then()` (or `.catch()`) throws a value, the Promise that was the return value of the `.then()` call is rejected with that thrown value. In this case, the code in `c1` that raises a TypeError causes `p2` to be rejected with that TypeError object. Since we did not specify an error handler for `p2`, `p3` will be rejected as well. `c2` will not be called, and the TypeError will be passed to `c3`, which has code to explicitly check for and handle this type of error.

There are a couple of things worth noting about this code. First, notice that the error object thrown with a regular, synchronous `throw` statement ends up being handled asynchronously with a `.catch()` method invocation in a Promise chain. This should make it clear why this shorthand method is preferred over passing a second argument to `.then()`, and also why it is so idiomatic to end Promise chains with a `.catch()` call.

Before we leave the topic of error handling, I want to point out that, although it is idiomatic to end every Promise chain with a `.catch()` to clean up (or at least log) any errors that occurred in the chain, it is also perfectly valid to use `.catch()` elsewhere in a Promise chain. If one of the stages in your Promise chain can fail with an error, and if the error is some kind of recoverable error that should not stop the rest of the chain from running, then you can insert a `.catch()` call in the chain, resulting in code that might look like this:

```
startAsyncOperation()
    .then(doStageTwo)
    .catch(recoverFromStageTwoError)
    .then(doStageThree)
    .then(doStageFour)
    .catch(logStageThreeAndFourErrors);
```

Remember that the callback you pass to `.catch()` will only be invoked if the callback at a previous stage throws an error. If the callback returns normally, then the `.catch()` callback will be skipped, and the return value of the previous callback will become the input to the next `.then()` callback. Also remember that `.catch()` callbacks are not just for reporting errors, but for handling and recovering from errors. Once an error has been passed to a `.catch()` callback, it stops propagating down the Promise chain. A `.catch()` callback can throw a new error, but if it returns normally, than that return value is used to resolve and/or fulfill the associated Promise, and the error stops propagating.

Let's be concrete about this: in the preceding code example, if either `startAsyncOperation()` or `doStageTwo()` throws an error, then the `recoverFromStageTwoError()` function will be invoked. If `recoverFromStageTwoError()` returns normally, then its return value will be passed to `doStageThree()` and the asynchronous operation continues normally. On the other hand, if `recoverFromStageTwoError()` was unable to recover, it will itself throw an error (or it will rethrow the error that it was passed). In this case, neither `doStageThree()` nor `doStageFour()` will be invoked, and the error thrown by `recoverFromStageTwoError()` would be passed to `logStageThreeAndFourErrors()`.

Sometimes, in complex network environments, errors can occur more or less at random, and it can be appropriate to handle those errors by simply retrying the asynchronous request. Imagine you've written a Promise-based operation to query a database:

```
queryDatabase()
    .then(displayTable)
    .catch(displayDatabaseError);
```

Now suppose that transient network load issues are causing this to fail about 1% of the time. A simple solution might be to retry the query with a `.catch()` call:

```
queryDatabase()
    .catch(e => wait(500).then(queryDatabase))  // On failure, wait and retry
    .then(displayTable)
    .catch(displayDatabaseError);
```

If the hypothetical failures are truly random, then adding this one line of code should reduce your error rate from 1% to .01%.

Returning from a Promise Callback

Let's return one last time to the earlier URL-fetching example, and consider the `c1` callback that we passed to the first `.then()` invocation. Notice that there are three ways that `c1` can terminate. It can return normally with the Promise returned by the `.json()` call. This causes `p2` to be resolved, but whether that Promise is fulfilled or

rejected depends on what happens with the newly returned Promise. `c1` can also return normally with the value `null`, which causes `p2` to be fulfilled immediately. Finally, `c1` can terminate by throwing an error, which causes `p2` to be rejected. These are the three possible outcomes for a Promise, and the code in `c1` demonstrates how the callback can cause each outcome.

In a Promise chain, the value returned (or thrown) at one stage of the chain becomes the input to the next stage of the chain, so it is critical to get this right. In practice, forgetting to return a value from a callback function is actually a common source of Promise-related bugs, and this is exacerbated by JavaScript's arrow function shortcut syntax. Consider this line of code that we saw earlier:

```
.catch(e => wait(500).then(queryDatabase))
```

Recall from Chapter 8 that arrow functions allow a lot of shortcuts. Since there is exactly one argument (the error value), we can omit the parentheses. Since the body of the function is a single expression, we can omit the curly braces around the function body, and the value of the expression becomes the return value of the function. Because of these shortcuts, the preceding code is correct. But consider this innocuous-seeming change:

```
.catch(e => { wait(500).then(queryDatabase) })
```

By adding the curly braces, we no longer get the automatic return. This function now returns `undefined` instead of returning a Promise, which means that the next stage in this Promise chain will be invoked with `undefined` as its input rather than the result of the retried query. It is a subtle error that may not be easy to debug.

13.2.5 Promises in Parallel

We've spent a lot of time talking about Promise chains for sequentially running the asynchronous steps of a larger asynchronous operation. Sometimes, though, we want to execute a number of asynchronous operations in parallel. The function `Promise.all()` can do this. `Promise.all()` takes an array of Promise objects as its input and returns a Promise. The returned Promise will be rejected if any of the input Promises are rejected. Otherwise, it will be fulfilled with an array of the fulfillment values of each of the input Promises. So, for example, if you want to fetch the text content of multiple URLs, you could use code like this:

```
// We start with an array of URLs
const urls = [ /* zero or more URLs here */ ];
// And convert it to an array of Promise objects
promises = urls.map(url => fetch(url).then(r => r.text()));
// Now get a Promise to run all those Promises in parallel
Promise.all(promises)
    .then(bodies => { /* do something with the array of strings */ })
    .catch(e => console.error(e));
```

`Promise.all()` is slightly more flexible than described before. The input array can contain both Promise objects and non-Promise values. If an element of the array is not a Promise, it is treated as if it is the value of an already fulfilled Promise and is simply copied unchanged into the output array.

The Promise returned by `Promise.all()` rejects when any of the input Promises is rejected. This happens immediately upon the first rejection and can happen while other input Promises are still pending. In ES2020, `Promise.allSettled()` takes an array of input Promises and returns a Promise, just like `Promise.all()` does. But `Promise.allSettled()` never rejects the returned Promise, and it does not fulfill that Promise until all of the input Promises have settled. The Promise resolves to an array of objects, with one object for each input Promise. Each of these returned objects has a `status` property set to "fulfilled" or "rejected." If the status is "fulfilled", then the object will also have a `value` property that gives the fulfillment value. And if the status is "rejected", then the object will also have a `reason` property that gives the error or rejection value of the corresponding Promise:

```
Promise.allSettled([Promise.resolve(1), Promise.reject(2), 3]).then(results => {
    results[0]  // => { status: "fulfilled", value: 1 }
    results[1]  // => { status: "rejected", reason: 2 }
    results[2]  // => { status: "fulfilled", value: 3 }
});
```

Occasionally, you may want to run a number of Promises at once but may only care about the value of the first one to fulfill. In that case, you can use `Promise.race()` instead of `Promise.all()`. It returns a Promise that is fulfilled or rejected when the first of the Promises in the input array is fulfilled or rejected. (Or, if there are any non-Promise values in the input array, it simply returns the first of those.)

13.2.6 Making Promises

We've used the Promise-returning function `fetch()` in many of the previous examples because it is one of the simplest functions built in to web browsers that returns a Promise. Our discussion of Promises has also relied on hypothetical Promise-returning functions `getJSON()` and `wait()`. Functions written to return Promises really are quite useful, and this section shows how you can create your own Promise-based APIs. In particular, we'll show implementations of `getJSON()` and `wait()`.

Promises based on other Promises

It is easy to write a function that returns a Promise if you have some other Promise-returning function to start with. Given a Promise, you can always create (and return) a new one by calling `.then()`. So if we use the existing `fetch()` function as a starting point, we can write `getJSON()` like this:

```
function getJSON(url) {
    return fetch(url).then(response => response.json());
}
```

The code is trivial because the Response object of the `fetch()` API has a predefined `json()` method. The `json()` method returns a Promise, which we return from our callback (the callback is an arrow function with a single-expression body, so the return is implicit), so the Promise returned by `getJSON()` resolves to the Promise returned by `response.json()`. When that Promise fulfills, the Promise returned by `getJSON()` fulfills to the same value. Note that there is no error handling in this `getJSON()` implementation. Instead of checking `response.ok` and the Content-Type header, we instead just allow the `json()` method to reject the Promise it returned with a SyntaxError if the response body cannot be parsed as JSON.

Let's write another Promise-returning function, this time using `getJSON()` as the source of the initial Promise:

```
function getHighScore() {
    return getJSON("/api/user/profile").then(profile => profile.highScore);
}
```

We're assuming that this function is part of some sort of web-based game and that the URL "/api/user/profile" returns a JSON-formatted data structure that includes a `highScore` property.

Promises based on synchronous values

Sometimes, you may need to implement an existing Promise-based API and return a Promise from a function, even though the computation to be performed does not actually require any asynchronous operations. In that case, the static methods `Promise.resolve()` and `Promise.reject()` will do what you want. `Promise.resolve()` takes a value as its single argument and returns a Promise that will immediately (but asynchronously) be fulfilled to that value. Similarly, `Promise.reject()` takes a single argument and returns a Promise that will be rejected with that value as the reason. (To be clear: the Promises returned by these static methods are not already fulfilled or rejected when they are returned, but they will fulfill or reject immediately after the current synchronous chunk of code has finished running. Typically, this happens within a few milliseconds unless there are many pending asynchronous tasks waiting to run.)

Recall from §13.2.3 that a resolved Promise is not the same thing as a fulfilled Promise. When we call `Promise.resolve()`, we typically pass the fulfillment value to create a Promise object that will very soon fulfill to that value. The method is not named `Promise.fulfill()`, however. If you pass a Promise `p1` to `Promise.resolve()`, it will return a new Promise `p2`, which is immediately resolved, but which will not be fulfilled or rejected until `p1` is fulfilled or rejected.

It is possible, but unusual, to write a Promise-based function where the value is computed synchronously and returned asynchronously with `Promise.resolve()`. It is fairly common, however, to have synchronous special cases within an asynchronous function, and you can handle these special cases with `Promise.resolve()` and `Promise.reject()`. In particular, if you detect error conditions (such as bad argument values) before beginning an asynchronous operation, you can report that error by returning a Promise created with `Promise.reject()`. (You could also just throw an error synchronously in that case, but that is considered poor form because then the caller of your function needs to write both a synchronous `catch` clause and use an asynchronous `.catch()` method to handle errors.) Finally, `Promise.resolve()` is sometimes useful to create the initial Promise in a chain of Promises. We'll see a couple of examples that use it this way.

Promises from scratch

For both `getJSON()` and `getHighScore()`, we started off by calling an existing function to get an initial Promise, and created and returned a new Promise by calling the `.then()` method of that initial Promise. But what about writing a Promise-returning function when you can't use another Promise-returning function as the starting point? In that case, you use the `Promise()` constructor to create a new Promise object that you have complete control over. Here's how it works: you invoke the `Promise()` constructor and pass a function as its only argument. The function you pass should be written to expect two parameters, which, by convention, should be named `resolve` and `reject`. The constructor synchronously calls your function with function arguments for the `resolve` and `reject` parameters. After calling your function, the `Promise()` constructor returns the newly created Promise. That returned Promise is under the control of the function you passed to the constructor. That function should perform some asynchronous operation and then call the `resolve` function to resolve or fulfill the returned Promise or call the `reject` function to reject the returned Promise. Your function does not have to be asynchronous: it can call `resolve` or `reject` synchronously, but the Promise will still be resolved, fulfilled, or rejected asynchronously if you do this.

It can be hard to understand the functions passed to a function passed to a constructor by just reading about it, but hopefully some examples will make this clear. Here's how to write the Promise-based `wait()` function that we used in various examples earlier in the chapter:

```
function wait(duration) {
    // Create and return a new Promise
    return new Promise((resolve, reject) => { // These control the Promise
        // If the argument is invalid, reject the Promise
        if (duration < 0) {
            reject(new Error("Time travel not yet implemented"));
        }
```

```
            // Otherwise, wait asynchronously and then resolve the Promise.
            // setTimeout will invoke resolve() with no arguments, which means
            // that the Promise will fulfill with the undefined value.
            setTimeout(resolve, duration);
        });
    }
```

Note that the pair of functions that you use to control the fate of a Promise created with the `Promise()` constructor are named `resolve()` and `reject()`, not `fulfill()` and `reject()`. If you pass a Promise to `resolve()`, the returned Promise will resolve to that new Promise. Often, however, you will pass a non-Promise value, which fulfills the returned Promise with that value.

Example 13-1 is another example of using the `Promise()` constructor. This one implements our `getJSON()` function for use in Node, where the `fetch()` API is not built in. Remember that we started this chapter with a discussion of asynchronous callbacks and events. This example uses both callbacks and event handlers and is a good demonstration, therefore, of how we can implement Promise-based APIs on top of other styles of asynchronous programming.

Example 13-1. An asynchronous getJSON() function

```
const http = require("http");

function getJSON(url) {
    // Create and return a new Promise
    return new Promise((resolve, reject) => {
        // Start an HTTP GET request for the specified URL
        request = http.get(url, response => { // called when response starts
            // Reject the Promise if the HTTP status is wrong
            if (response.statusCode !== 200) {
                reject(new Error(`HTTP status ${response.statusCode}`));
                response.resume();  // so we don't leak memory
            }
            // And reject if the response headers are wrong
            else if (response.headers["content-type"] !== "application/json") {
                reject(new Error("Invalid content-type"));
                response.resume();  // don't leak memory
            }
            else {
                // Otherwise, register events to read the body of the response
                let body = "";
                response.setEncoding("utf-8");
                response.on("data", chunk => { body += chunk; });
                response.on("end", () => {
                    // When the response body is complete, try to parse it
                    try {
                        let parsed = JSON.parse(body);
                        // If it parsed successfully, fulfill the Promise
                        resolve(parsed);
```

```
                } catch(e) {
                    // If parsing failed, reject the Promise
                    reject(e);
                }
            });
        }
    });
    // We also reject the Promise if the request fails before we
    // even get a response (such as when the network is down)
    request.on("error", error => {
        reject(error);
    });
});
}
```

13.2.7 Promises in Sequence

`Promise.all()` makes it easy to run an arbitrary number of Promises in parallel. And Promise chains make it easy to express a sequence of a fixed number of Promises. Running an arbitrary number of Promises in sequence is trickier, however. Suppose, for example, that you have an array of URLs to fetch, but that to avoid overloading your network, you want to fetch them one at a time. If the array is of arbitrary length and unknown content, you can't write out a Promise chain in advance, so you need to build one dynamically, with code like this:

```
function fetchSequentially(urls) {
    // We'll store the URL bodies here as we fetch them
    const bodies = [];

    // Here's a Promise-returning function that fetches one body
    function fetchOne(url) {
        return fetch(url)
            .then(response => response.text())
            .then(body => {
                // We save the body to the array, and we're purposely
                // omitting a return value here (returning undefined)
                bodies.push(body);
            });
    }

    // Start with a Promise that will fulfill right away (with value undefined)
    let p = Promise.resolve(undefined);

    // Now loop through the desired URLs, building a Promise chain
    // of arbitrary length, fetching one URL at each stage of the chain
    for(url of urls) {
        p = p.then(() => fetchOne(url));
    }

    // When the last Promise in that chain is fulfilled, then the
    // bodies array is ready. So let's return a Promise for that
```

```
        // bodies array. Note that we don't include any error handlers:
        // we want to allow errors to propagate to the caller.
        return p.then(() => bodies);
    }
```

With this `fetchSequentially()` function defined, we could fetch the URLs one at a time with code much like the fetch-in-parallel code we used earlier to demonstrate `Promise.all()`:

```
fetchSequentially(urls)
    .then(bodies => { /* do something with the array of strings */ })
    .catch(e => console.error(e));
```

The `fetchSequentially()` function starts by creating a Promise that will fulfill immediately after it returns. It then builds a long, linear Promise chain off of that initial Promise and returns the last Promise in the chain. It is like setting up a row of dominoes and then knocking the first one over.

There is another (possibly more elegant) approach that we can take. Rather than creating the Promises in advance, we can have the callback for each Promise create and return the next Promise. That is, instead of creating and chaining a bunch of Promises, we instead create Promises that resolve to other Promises. Rather than creating a domino-like chain of Promises, we are instead creating a sequence of Promises nested one inside the other like a set of matryoshka dolls. With this approach, our code can return the first (outermost) Promise, knowing that it will eventually fulfill (or reject!) to the same value that the last (innermost) Promise in the sequence does. The `promiseSequence()` function that follows is written to be generic and is not specific to URL fetching. It is here at the end of our discussion of Promises because it is complicated. If you've read this chapter carefully, however, I hope you'll be able to understand how it works. In particular, note that the nested function inside `promiseSequence()` appears to call itself recursively, but because the "recursive" call is through a `then()` method, there is not actually any traditional recursion happening:

```
// This function takes an array of input values and a "promiseMaker" function.
// For any input value x in the array, promiseMaker(x) should return a Promise
// that will fulfill to an output value. This function returns a Promise
// that fulfills to an array of the computed output values.
//
// Rather than creating the Promises all at once and letting them run in
// parallel, however, promiseSequence() only runs one Promise at a time
// and does not call promiseMaker() for a value until the previous Promise
// has fulfilled.
function promiseSequence(inputs, promiseMaker) {
    // Make a private copy of the array that we can modify
    inputs = [...inputs];

    // Here's the function that we'll use as a Promise callback
    // This is the pseudorecursive magic that makes this all work.
    function handleNextInput(outputs) {
```

```
            if (inputs.length === 0) {
                // If there are no more inputs left, then return the array
                // of outputs, finally fulfilling this Promise and all the
                // previous resolved-but-not-fulfilled Promises.
                return outputs;
            } else {
                // If there are still input values to process, then we'll
                // return a Promise object, resolving the current Promise
                // with the future value from a new Promise.
                let nextInput = inputs.shift(); // Get the next input value,
                return promiseMaker(nextInput)  // compute the next output value,
                    // Then create a new outputs array with the new output value
                    .then(output => outputs.concat(output))
                    // Then "recurse", passing the new, longer, outputs array
                    .then(handleNextInput);
            }
        }

        // Start with a Promise that fulfills to an empty array and use
        // the function above as its callback.
        return Promise.resolve([]).then(handleNextInput);
    }
```

This `promiseSequence()` function is intentionally generic. We can use it to fetch URLs with code like this:

```
// Given a URL, return a Promise that fulfills to the URL body text
function fetchBody(url) { return fetch(url).then(r => r.text()); }
// Use it to sequentially fetch a bunch of URL bodies
promiseSequence(urls, fetchBody)
    .then(bodies => { /* do something with the array of strings */ })
    .catch(console.error);
```

13.3 async and await

ES2017 introduces two new keywords—`async` and `await`—that represent a paradigm shift in asynchronous JavaScript programming. These new keywords dramatically simplify the use of Promises and allow us to write Promise-based, asynchronous code that looks like synchronous code that blocks while waiting for network responses or other asynchronous events. Although it is still important to understand how Promises work, much of their complexity (and sometimes even their very presence!) vanishes when you use them with `async` and `await`.

As we discussed earlier in the chapter, asynchronous code can't return a value or throw an exception the way that regular synchronous code can. And this is why Promises are designed the way the are. The value of a fulfilled Promise is like the return value of a synchronous function. And the value of a rejected Promise is like a value thrown by a synchronous function. This latter similarity is made explicit by the naming of the `.catch()` method. `async` and `await` take efficient, Promise-based code

and hide the Promises so that your asynchronous code can be as easy to read and as easy to reason about as inefficient, blocking, synchronous code.

13.3.1 await Expressions

The `await` keyword takes a Promise and turns it back into a return value or a thrown exception. Given a Promise object `p`, the expression `await p` waits until `p` settles. If `p` fulfills, then the value of `await p` is the fulfillment value of `p`. On the other hand, if `p` is rejected, then the `await p` expression throws the rejection value of `p`. We don't usually use `await` with a variable that holds a Promise; instead, we use it before the invocation of a function that returns a Promise:

```
let response = await fetch("/api/user/profile");
let profile = await response.json();
```

It is critical to understand right away that the `await` keyword does not cause your program to block and literally do nothing until the specified Promise settles. The code remains asynchronous, and the `await` simply disguises this fact. This means that *any code that uses* `await` *is itself asynchronous*.

13.3.2 async Functions

Because any code that uses `await` is asynchronous, there is one critical rule: *you can only use the* `await` *keyword within functions that have been declared with the* `async` *keyword*. Here's a version of the `getHighScore()` function from earlier in the chapter, rewritten to use `async` and `await`:

```
async function getHighScore() {
    let response = await fetch("/api/user/profile");
    let profile = await response.json();
    return profile.highScore;
}
```

Declaring a function `async` means that the return value of the function will be a Promise even if no Promise-related code appears in the body of the function. If an `async` function appears to return normally, then the Promise object that is the real return value of the function will resolve to that apparent return value. And if an `async` function appears to throw an exception, then the Promise object that it returns will be rejected with that exception.

The `getHighScore()` function is declared `async`, so it returns a Promise. And because it returns a Promise, we can use the `await` keyword with it:

```
displayHighScore(await getHighScore());
```

But remember, that line of code will only work if it is inside another `async` function! You can nest `await` expressions within `async` functions as deeply as you want. But if

you're at the top level[2] or are inside a function that is not `async` for some reason, then you can't use `await` and have to deal with a returned Promise in the regular way:

```
getHighScore().then(displayHighScore).catch(console.error);
```

You can use the `async` keyword with any kind of function. It works with the `function` keyword as a statement or as an expression. It works with arrow functions and with the method shortcut form in classes and object literals. (See Chapter 8 for more about the various ways to write functions.)

13.3.3 Awaiting Multiple Promises

Suppose that we've written our `getJSON()` function using `async`:

```
async function getJSON(url) {
    let response = await fetch(url);
    let body = await response.json();
    return body;
}
```

And now suppose that we want to fetch two JSON values with this function:

```
let value1 = await getJSON(url1);
let value2 = await getJSON(url2);
```

The problem with this code is that it is unnecessarily sequential: the fetch of the second URL will not begin until the first fetch is complete. If the second URL does not depend on the value obtained from the first URL, then we should probably try to fetch the two values at the same time. This is a case where the Promise-based nature of `async` functions shows. In order to await a set of concurrently executing `async` functions, we use `Promise.all()` just as we would if working with Promises directly:

```
let [value1, value2] = await Promise.all([getJSON(url1), getJSON(url2)]);
```

13.3.4 Implementation Details

Finally, in order to understand how `async` functions work, it may help to think about what is going on under the hood.

Suppose you write an `async` function like this:

```
async function f(x) { /* body */ }
```

You can think about this as a Promise-returning function wrapped around the body of your original function:

2 You can typically use `await` at the top level in a browser's developer console. And there is a pending proposal to allow top-level `await` in a future version of JavaScript.

```
function f(x) {
    return new Promise(function(resolve, reject) {
        try {
            resolve((function(x) { /* body */ })(x));
        }
        catch(e) {
            reject(e);
        }
    });
}
```

It is harder to express the `await` keyword in terms of a syntax transformation like this one. But think of the `await` keyword as a marker that breaks a function body up into separate, synchronous chunks. An ES2017 interpreter can break the function body up into a sequence of separate subfunctions, each of which gets passed to the `then()` method of the `await`-marked Promise that precedes it.

13.4 Asynchronous Iteration

We began this chapter with a discussion of callback- and event-based asynchrony, and when we introduced Promises, we noted that they were useful for single-shot asynchronous computations but were not suitable for use with sources of repetitive asynchronous events, such as `setInterval()`, the "click" event in a web browser, or the "data" event on a Node stream. Because single Promises do not work for sequences of asynchronous events, we also cannot use regular `async` functions and the `await` statements for these things.

ES2018 provides a solution, however. Asynchronous iterators are like the iterators described in Chapter 12, but they are Promise-based and are meant to be used with a new form of the `for/of` loop: `for/await`.

13.4.1 The for/await Loop

Node 12 makes its readable streams asynchronously iterable. This means you can read successive chunks of data from a stream with a `for/await` loop like this one:

```
const fs = require("fs");

async function parseFile(filename) {
    let stream = fs.createReadStream(filename, { encoding: "utf-8"});
    for await (let chunk of stream) {
        parseChunk(chunk); // Assume parseChunk() is defined elsewhere
    }
}
```

Like a regular `await` expression, the `for/await` loop is Promise-based. Roughly speaking, the asynchronous iterator produces a Promise and the `for/await` loop waits for that Promise to fulfill, assigns the fulfillment value to the loop variable, and

runs the body of the loop. And then it starts over, getting another Promise from the iterator and waiting for that new Promise to fulfill.

Suppose you have an array of URLs:

```
const urls = [url1, url2, url3];
```

You can call `fetch()` on each URL to get an array of Promises:

```
const promises = urls.map(url => fetch(url));
```

We saw earlier in the chapter that we could now use `Promise.all()` to wait for all the Promises in the array to be fulfilled. But suppose we want the results of the first fetch as soon as they become available and don't want to wait for all the URLs to be fetched. (Of course, the first fetch might take longer than any of the others, so this is not necessarily faster than using `Promise.all()`.) Arrays are iterable, so we can iterate through the array of promises with a regular `for/of` loop:

```
for(const promise of promises) {
    response = await promise;
    handle(response);
}
```

This example code uses a regular `for/of` loop with a regular iterator. But because this iterator returns Promises, we can also use the new `for/await` for slightly simpler code:

```
for await (const response of promises) {
    handle(response);
}
```

In this case, the `for/await` loop just builds the `await` call into the loop and makes our code slightly more compact, but the two examples do exactly the same thing. Importantly, both examples will only work if they are within functions declared `async`; a `for/await` loop is no different than a regular `await` expression in that way.

It is important to realize, however, that we're using `for/await` with a regular iterator in this example. Things are more interesting with fully asynchronous iterators.

13.4.2 Asynchronous Iterators

Let's review some terminology from Chapter 12. An *iterable* object is one that can be used with a `for/of` loop. It defines a method with the symbolic name `Symbol.iterator`. This method returns an *iterator* object. The iterator object has a `next()` method, which can be called repeatedly to obtain the values of the iterable object. The `next()` method of the iterator object returns *iteration result* objects. The iteration result object has a `value` property and/or a `done` property.

Asynchronous iterators are quite similar to regular iterators, but there are two important differences. First, an asynchronously iterable object implements a method with the symbolic name `Symbol.asyncIterator` instead of `Symbol.iterator`. (As we saw earlier, `for/await` is compatible with regular iterable objects but it prefers asynchronously iterable objects, and tries the `Symbol.asyncIterator` method before it tries the `Symbol.iterator` method.) Second, the `next()` method of an asynchronous iterator returns a Promise that resolves to an iterator result object instead of returning an iterator result object directly.

In the previous section, when we used `for/await` on a regular, synchronously iterable array of Promises, we were working with synchronous iterator result objects in which the `value` property was a Promise object but the `done` property was synchronous. True asynchronous iterators return Promises for iteration result objects, and both the `value` and the `done` properties are asynchronous. The difference is a subtle one: with asynchronous iterators, the choice about when iteration ends can be made asynchronously.

13.4.3 Asynchronous Generators

As we saw in Chapter 12, the easiest way to implement an iterator is often to use a generator. The same is true for asynchronous iterators, which we can implement with generator functions that we declare `async`. An async generator has the features of async functions and the features of generators: you can use `await` as you would in a regular async function, and you can use `yield` as you would in a regular generator. But values that you `yield` are automatically wrapped in Promises. Even the syntax for async generators is a combination: `async function` and `function *` combine into `async function *`. Here is an example that shows how you might use an async generator and a `for/await` loop to repetitively run code at fixed intervals using loop syntax instead of a `setInterval()` callback function:

```
// A Promise-based wrapper around setTimeout() that we can use await with.
// Returns a Promise that fulfills in the specified number of milliseconds
function elapsedTime(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
}

// An async generator function that increments a counter and yields it
// a specified (or infinite) number of times at a specified interval.
async function* clock(interval, max=Infinity) {
    for(let count = 1; count <= max; count++) { // regular for loop
        await elapsedTime(interval);            // wait for time to pass
        yield count;                            // yield the counter
    }
}
```

```
// A test function that uses the async generator with for/await
async function test() {                       // Async so we can use for/await
    for await (let tick of clock(300, 100)) { // Loop 100 times every 300ms
        console.log(tick);
    }
}
```

13.4.4 Implementing Asynchronous Iterators

Instead of using async generators to implement asynchronous iterators, it is also possible to implement them directly by defining an object with a `Symbol.asyncIterator()` method that returns an object with a `next()` method that returns a Promise that resolves to an iterator result object. In the following code, we re-implement the `clock()` function from the preceding example so that it is not a generator and instead just returns an asynchronously iterable object. Notice that the `next()` method in this example does not explicitly return a Promise; instead, we just declare `next()` to be async:

```
function clock(interval, max=Infinity) {
    // A Promise-ified version of setTimeout that we can use await with.
    // Note that this takes an absolute time instead of an interval.
    function until(time) {
        return new Promise(resolve => setTimeout(resolve, time - Date.now()));
    }

    // Return an asynchronously iterable object
    return {
        startTime: Date.now(),  // Remember when we started
        count: 1,               // Remember which iteration we're on
        async next() {          // The next() method makes this an iterator
            if (this.count > max) {     // Are we done?
                return { done: true };  // Iteration result indicating done
            }
            // Figure out when the next iteration should begin,
            let targetTime = this.startTime + this.count * interval;
            // wait until that time,
            await until(targetTime);
            // and return the count value in an iteration result object.
            return { value: this.count++ };
        },
        // This method means that this iterator object is also an iterable.
        [Symbol.asyncIterator]() { return this; }
    };
}
```

This iterator-based version of the `clock()` function fixes a flaw in the generator-based version. Note that, in this newer code, we target the absolute time at which each iteration should begin and subtract the current time from that in order to compute the interval that we pass to `setTimeout()`. If we use `clock()` with a `for/await`

loop, this version will run loop iterations more precisely at the specified interval because it accounts for the time required to actually run the body of the loop. But this fix isn't just about timing accuracy. The `for/await` loop always waits for the Promise returned by one iteration to be fulfilled before it begins the next iteration. But if you use an asynchronous iterator without a `for/await` loop, there is nothing to prevent you from calling the `next()` method whenever you want. With the generator-based version of `clock()`, if you call the `next()` method three times sequentially, you'll get three Promises that will all fulfill at almost exactly the same time, which is probably not what you want. The iterator-based version we've implemented here does not have that problem.

The benefit of asynchronous iterators is that they allow us to represent streams of asynchronous events or data. The `clock()` function discussed previously was fairly simple to write because the source of the asynchrony was the `setTimeout()` calls we were making ourselves. But when we are trying to work with other asynchronous sources, such as the triggering of event handlers, it becomes substantially harder to implement asynchronous iterators—we typically have a single event handler function that responds to events, but each call to the iterator's `next()` method must return a distinct Promise object, and multiple calls to `next()` may occur before the first Promise resolves. This means that any asynchronous iterator method must be able to maintain an internal queue of Promises that it resolves in order as asynchronous events are occurring. If we encapsulate this Promise-queueing behavior into an AsyncQueue class, then it becomes much easier to write asynchronous iterators based on AsyncQueue.[3]

The AsyncQueue class that follows has `enqueue()` and `dequeue()` methods as you'd expect for a queue class. The `dequeue()` method returns a Promise rather than an actual value, however, which means that it is OK to call `dequeue()` before `enqueue()` has ever been called. The AsyncQueue class is also an asynchronous iterator, and is intended to be used with a `for/await` loop whose body runs once each time a new value is asynchronously enqueued. (AsyncQueue has a `close()` method. Once called, no more values can be enqueued. When a closed queue is empty, the `for/await` loop will stop looping.)

Note that the implementation of AsyncQueue does not use `async` or `await` and instead works directly with Promises. The code is somewhat complicated, and you can use it to test your understanding of the material we've covered in this long chapter. Even if you don't fully understand the AsyncQueue implementation, do take a

3 I learned about this approach to asynchronous iteration from the blog of Dr. Axel Rauschmayer, *https://2ality.com*.

look at the shorter example that follows it: it implements a simple but very interesting asynchronous iterator on top of AsyncQueue.

```
/**
 * An asynchronously iterable queue class. Add values with enqueue()
 * and remove them with dequeue(). dequeue() returns a Promise, which
 * means that values can be dequeued before they are enqueued. The
 * class implements [Symbol.asyncIterator] and next() so that it can
 * be used with the for/await loop (which will not terminate until
 * the close() method is called.)
 */
class AsyncQueue {
    constructor() {
        // Values that have been queued but not dequeued yet are stored here
        this.values = [];
        // When Promises are dequeued before their corresponding values are
        // queued, the resolve methods for those Promises are stored here.
        this.resolvers = [];
        // Once closed, no more values can be enqueued, and no more unfulfilled
        // Promises returned.
        this.closed = false;
    }

    enqueue(value) {
        if (this.closed) {
            throw new Error("AsyncQueue closed");
        }
        if (this.resolvers.length > 0) {
            // If this value has already been promised, resolve that Promise
            const resolve = this.resolvers.shift();
            resolve(value);
        }
        else {
            // Otherwise, queue it up
            this.values.push(value);
        }
    }

    dequeue() {
        if (this.values.length > 0) {
            // If there is a queued value, return a resolved Promise for it
            const value = this.values.shift();
            return Promise.resolve(value);
        }
        else if (this.closed) {
            // If no queued values and we're closed, return a resolved
            // Promise for the "end-of-stream" marker
            return Promise.resolve(AsyncQueue.EOS);
        }
        else {
            // Otherwise, return an unresolved Promise,
            // queuing the resolver function for later use
```

```
            return new Promise((resolve) => { this.resolvers.push(resolve); });
        }
    }

    close() {
        // Once the queue is closed, no more values will be enqueued.
        // So resolve any pending Promises with the end-of-stream marker
        while(this.resolvers.length > 0) {
            this.resolvers.shift()(AsyncQueue.EOS);
        }
        this.closed = true;
    }

    // Define the method that makes this class asynchronously iterable
    [Symbol.asyncIterator]() { return this; }

    // Define the method that makes this an asynchronous iterator. The
    // dequeue() Promise resolves to a value or the EOS sentinel if we're
    // closed. Here, we need to return a Promise that resolves to an
    // iterator result object.
    next() {
        return this.dequeue().then(value => (value === AsyncQueue.EOS)
                                   ? { value: undefined, done: true }
                                   : { value: value, done: false });
    }
}

// A sentinel value returned by dequeue() to mark "end of stream" when closed
AsyncQueue.EOS = Symbol("end-of-stream");
```

Because this AsyncQueue class defines the asynchronous iteration basics, we can create our own, more interesting asynchronous iterators simply by asynchronously queueing values. Here's an example that uses AsyncQueue to produce a stream of web browser events that can be handled with a `for/await` loop:

```
// Push events of the specified type on the specified document element
// onto an AsyncQueue object, and return the queue for use as an event stream
function eventStream(elt, type) {
    const q = new AsyncQueue();                 // Create a queue
    elt.addEventListener(type, e=>q.enqueue(e)); // Enqueue events
    return q;
}

async function handleKeys() {
    // Get a stream of keypress events and loop once for each one
    for await (const event of eventStream(document, "keypress")) {
        console.log(event.key);
    }
}
```

13.5 Summary

In this chapter, you have learned:

- Most real-world JavaScript programming is asynchronous.
- Traditionally, asynchrony has been handled with events and callback functions. This can get complicated, however, because you can end up with multiple levels of callbacks nested inside other callbacks, and because it is difficult to do robust error handling.
- Promises provide a new way of structuring callback functions. If used correctly (and unfortunately, Promises are easy to use incorrectly), they can convert asynchronous code that would have been nested into linear chains of `then()` calls where one asynchronous step of a computation follows another. Also, Promises allow you to centralize your error-handling code into a single `catch()` call at the end of a chain of `then()` calls.
- The `async` and `await` keywords allow us to write asynchronous code that is Promise-based under the hood but that looks like synchronous code. This makes the code easier to understand and reason about. If a function is declared `async`, it will implicitly return a Promise. Inside an `async` function, you can `await` a Promise (or a function that returns a Promise) as if the Promise value was synchronously computed.
- Objects that are asynchronously iterable can be used with a `for/await` loop. You can create asynchronously iterable objects by implementing a `[Symbol.asyncIterator]()` method or by invoking an `async function *` generator function. Asynchronous iterators provide an alternative to "data" events on streams in Node and can be used to represent a stream of user input events in client-side JavaScript.

CHAPTER 14
Metaprogramming

This chapter covers a number of advanced JavaScript features that are not commonly used in day-to-day programming but that may be valuable to programmers writing reusable libraries and of interest to anyone who wants to tinker with the details about how JavaScript objects behave.

Many of the features described here can loosely be described as "metaprogramming": if regular programming is writing code to manipulate data, then metaprogramming is writing code to manipulate other code. In a dynamic language like JavaScript, the lines between programming and metaprogramming are blurry—even the simple ability to iterate over the properties of an object with a `for/in` loop might be considered "meta" by programmers accustomed to more static languages.

The metaprogramming topics covered in this chapter include:

- §14.1 Controlling the enumerability, deleteability, and configurability of object properties
- §14.2 Controlling the extensibility of objects, and creating "sealed" and "frozen" objects
- §14.3 Querying and setting the prototypes of objects
- §14.4 Fine-tuning the behavior of your types with well-known Symbols
- §14.5 Creating DSLs (domain-specific languages) with template tag functions
- §14.6 Probing objects with `reflect` methods
- §14.7 Controlling object behavior with Proxy

14.1 Property Attributes

The properties of a JavaScript object have names and values, of course, but each property also has three associated attributes that specify how that property behaves and what you can do with it:

- The *writable* attribute specifies whether or not the value of a property can change.
- The *enumerable* attribute specifies whether the property is enumerated by the `for/in` loop and the `Object.keys()` method.
- The *configurable* attribute specifies whether a property can be deleted and also whether the property's attributes can be changed.

Properties defined in object literals or by ordinary assignment to an object are writable, enumerable, and configurable. But many of the properties defined by the JavaScript standard library are not.

This section explains the 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.

Recall from §6.10.6 that, while "data properties" have a value, "accessor properties" have a getter and/or a setter method instead. For the purposes of this section, we are going to consider the 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 JavaScript 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.

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");

// Here is an object with a read-only accessor property
const random = {
    get octet() { return Math.floor(Math.random()*256); },
};

// 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 §14.3); see also the similar `Reflect.getOwnPropertyDescriptor()` function in §14.6.)

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:

```
let o = {};  // Start with no properties at all
// Add a non-enumerable 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 non-enumerable
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. See also the very similar function `Reflect.defineProperty()` in §14.6.

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:

```
let p = Object.defineProperties({}, {
    x: { value: 1, writable: true, enumerable: true, configurable: true },
    y: { value: 1, writable: true, enumerable: true, configurable: true },
    r: {
        get() { return Math.sqrt(this.x*this.x + this.y*this.y); },
        enumerable: true,
        configurable: true
    }
});
p.r  // => Math.SQRT2
```

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()`).

The `Object.create()` method was introduced in §6.2. 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 non-extensible (see §14.2) 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 a 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).

§6.7 described the `Object.assign()` function that copies property values from one or more source objects into a target object. `Object.assign()` only copies enumerable properties, and property values, not property attributes. This is normally what we want, but it does mean, for example, that if one of the source objects has an accessor property, it is the value returned by the getter function that is copied to the target object, not the getter function itself. Example 14-1 demonstrates how we can use `Object.getOwnPropertyDescriptor()` and `Object.defineProperty()` to create a variant of `Object.assign()` that copies entire property descriptors rather than just copying property values.

Example 14-1. Copying properties and their attributes from one object to another

```
/*
 * Define a new Object.assignDescriptors() function that works like
 * Object.assign() except that it copies property descriptors from
 * source objects into the target object instead of just copying
 * property values. This function copies all own properties, both
 * enumerable and non-enumerable. And because it copies descriptors,
 * it copies getter functions from source objects and overwrites setter
 * functions in the target object rather than invoking those getters and
 * setters.
 *
 * Object.assignDescriptors() propagates any TypeErrors thrown by
 * Object.defineProperty(). This can occur if the target object is sealed
 * or frozen or if any of the source properties try to change an existing
 * non-configurable property on the target object.
```

```
 *
 * Note that the assignDescriptors property is added to Object with
 * Object.defineProperty() so that the new function can be created as
 * a non-enumerable property like Object.assign().
 */
Object.defineProperty(Object, "assignDescriptors", {
    // Match the attributes of Object.assign()
    writable: true,
    enumerable: false,
    configurable: true,
    // The function that is the value of the assignDescriptors property.
    value: function(target, ...sources) {
        for(let source of sources) {
            for(let name of Object.getOwnPropertyNames(source)) {
                let desc = Object.getOwnPropertyDescriptor(source, name);
                Object.defineProperty(target, name, desc);
            }

            for(let symbol of Object.getOwnPropertySymbols(source)) {
                let desc = Object.getOwnPropertyDescriptor(source, symbol);
                Object.defineProperty(target, symbol, desc);
            }
        }
        return target;
    }
});

let o = {c: 1, get count() {return this.c++;}}; // Define object with getter
let p = Object.assign({}, o);                   // Copy the property values
let q = Object.assignDescriptors({}, o);        // Copy the property descriptors
p.count   // => 1: This is now just a data property so
p.count   // => 1: ...the counter does not increment.
q.count   // => 2: Incremented once when we copied it the first time,
q.count   // => 3: ...but we copied the getter method so it increments.
```

14.2 Object Extensibility

The *extensible* attribute of an object specifies whether new properties can be added to the object or not. Ordinary JavaScript objects are extensible by default, but you can change that with the functions described in this section.

To determine whether an object is extensible, pass it to `Object.isExtensible()`. To make an object non-extensible, pass it to `Object.preventExtensions()`. Once you have done this, any attempt to add a new property to the object will throw a TypeError in strict mode and simply fail silently without an error in non-strict mode. In addition, attempting to change the prototype (see §14.3) of a non-extensible object will always throw a TypeError.

Note that there is no way to make an object extensible again once you have made it non-extensible. Also note that calling `Object.preventExtensions()` only affects the

extensibility of the object itself. If new properties are added to the prototype of a non-extensible object, the non-extensible object will inherit those new properties.

Two similar functions, `Reflect.isExtensible()` and `Reflect.preventExtensions()`, are described in §14.6.

The purpose of the *extensible* attribute is to be able to "lock down" objects into a known state and prevent outside tampering. The *extensible* attribute of objects is often used in conjunction with the *configurable* and *writable* attributes of properties, and JavaScript defines functions that make it easy to set these attributes together:

- `Object.seal()` works like `Object.preventExtensions()`, but in addition to making the object non-extensible, 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 non-extensible 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 non-enumerable property
let o = Object.seal(Object.create(Object.freeze({x: 1}),
                                  {y: {value: 2, writable: true}}));
```

If you are writing a JavaScript library that passes objects to callback functions written by the users of your library, you might use `Object.freeze()` on those objects to prevent the user's code from modifying them. This is easy and convenient to do, but there are trade-offs: frozen objects can interfere with common JavaScript testing strategies, for example.

14.3 The prototype Attribute

An object's `prototype` attribute specifies the object from which it inherits properties. (Review §6.2.3 and §6.3.2 for more on prototypes and property inheritance.) This is such an important attribute that we usually simply say "the prototype of `o`" rather than "the `prototype` attribute of `o`." Remember also that when `prototype` appears in code font, it refers to an ordinary object property, not to the `prototype` attribute: Chapter 9 explained that the `prototype` property of a constructor function specifies the `prototype` attribute of the objects created with that constructor.

The `prototype` attribute is set when an object is created. 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.

You can query the prototype of any object by passing that object to `Object.getPrototypeOf()`:

```
Object.getPrototypeOf({})      // => Object.prototype
Object.getPrototypeOf([])      // => Array.prototype
Object.getPrototypeOf(()=>{})  // => Function.prototype
```

A very similar function, `Reflect.getPrototypeOf()`, is described in §14.6.

To determine whether one object is the prototype of (or is part of the prototype chain of) another object, use the `isPrototypeOf()` method:

```
let p = {x: 1};                  // Define a prototype object.
let 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
Object.prototype.isPrototypeOf(o) // => true: o does too
```

Note that `isPrototypeOf()` performs a function similar to the `instanceof` operator (see §4.9.4).

The `prototype` attribute of an object is set when the object is created and normally remains fixed. You can, however, change the prototype of an object with `Object.setPrototypeOf()`:

```
let o = {x: 1};
let p = {y: 2};
Object.setPrototypeOf(o, p); // Set the prototype of o to p
o.y      // => 2: o now inherits the property y
let a = [1, 2, 3];
Object.setPrototypeOf(a, p); // Set the prototype of array a to p
a.join   // => undefined: a no longer has a join() method
```

There is generally no need to ever use `Object.setPrototypeOf()`. JavaScript implementations may make aggressive optimizations based on the assumption that the prototype of an object is fixed and unchanging. This means that if you ever call `Object.setPrototypeOf()`, any code that uses the altered objects may run much slower than it would normally.

A similar function, `Reflect.setPrototypeOf()`, is described in §14.6.

Some early browser implementations of JavaScript exposed the `prototype` attribute of an object through the `__proto__` property (written with two underscores at the start and end). This has long since been deprecated, but enough existing code on the web depends on `__proto__` that the ECMAScript standard mandates it for all JavaScript implementations that run in web browsers. (Node supports it, too, though the standard does not require it for Node.) In modern JavaScript, `__proto__` is readable and writeable, and you can (though you shouldn't) use it as an alternative to `Object.getPrototypeOf()` and `Object.setPrototypeOf()`. One interesting use of `__proto__`, however, is to define the prototype of an object literal:

```
let p = {z: 3};
let o = {
    x: 1,
    y: 2,
    __proto__: p
};
o.z  // => 3: o inherits from p
```

14.4 Well-Known Symbols

The Symbol type was added to JavaScript in ES6, and one of the primary reasons for doing so was to safely add extensions to the language without breaking compatibility with code already deployed on the web. We saw an example of this in Chapter 12, where we learned that you can make a class iterable by implementing a method whose "name" is the Symbol `Symbol.iterator`.

`Symbol.iterator` is the best-known example of the "well-known Symbols." These are a set of Symbol values stored as properties of the `Symbol()` factory function that are used to allow JavaScript code to control certain low-level behaviors of objects and classes. The subsections that follow describe each of these well-known Symbols and explain how they can be used.

14.4.1 Symbol.iterator and Symbol.asyncIterator

The `Symbol.iterator` and `Symbol.asyncIterator` Symbols allow objects or classes to make themselves iterable or asynchronously iterable. They were covered in detail in Chapter 12 and §13.4.2, respectively, and are mentioned again here only for completeness.

14.4.2 Symbol.hasInstance

When the `instanceof` operator was described in §4.9.4, we said that the righthand side must be a constructor function and that the expression `o instanceof f` was evaluated by looking for the value `f.prototype` within the prototype chain of `o`. That is still true, but in ES6 and beyond, `Symbol.hasInstance` provides an alternative. In ES6, if the righthand side of `instanceof` is any object with a `[Symbol.hasInstance]` method, then that method is invoked with the lefthand side value as its argument, and the return value of the method, converted to a boolean, becomes the value of the `instanceof` operator. And, of course, if the value on the righthand side does not have a `[Symbol.hasInstance]` method but is a function, then the `instanceof` operator behaves in its ordinary way.

`Symbol.hasInstance` means that we can use the `instanceof` operator to do generic type checking with suitably defined pseudotype objects. For example:

```
// Define an object as a "type" we can use with instanceof
let uint8 = {
    [Symbol.hasInstance](x) {
        return Number.isInteger(x) && x >= 0 && x <= 255;
    }
};
128 instanceof uint8     // => true
256 instanceof uint8     // => false: too big
Math.PI instanceof uint8 // => false: not an integer
```

Note that this example is clever but confusing because it uses a nonclass object where a class would normally be expected. It would be just as easy—and clearer to readers of your code—to write a `isUint8()` function instead of relying on this `Symbol.hasInstance` behavior.

14.4.3 Symbol.toStringTag

If you invoke the `toString()` method of a basic JavaScript object, you get the string "[object Object]":

```
{}.toString()  // => "[object Object]"
```

If you invoke this same `Object.prototype.toString()` function as a method of instances of built-in types, you get some interesting results:

```
Object.prototype.toString.call([])     // => "[object Array]"
Object.prototype.toString.call(/./)    // => "[object RegExp]"
Object.prototype.toString.call(()=>{}) // => "[object Function]"
Object.prototype.toString.call("")     // => "[object String]"
Object.prototype.toString.call(0)      // => "[object Number]"
Object.prototype.toString.call(false)  // => "[object Boolean]"
```

It turns out that you can use this `Object.prototype.toString().call()` technique with any JavaScript value to obtain the "class attribute" of an object that contains type information that is not otherwise available. The following `classof()` function is arguably more useful than the `typeof` operator, which makes no distinction between types of objects:

```
function classof(o) {
    return Object.prototype.toString.call(o).slice(8,-1);
}

classof(null)       // => "Null"
classof(undefined)  // => "Undefined"
classof(1)          // => "Number"
classof(10n**100n)  // => "BigInt"
classof("")         // => "String"
classof(false)      // => "Boolean"
classof(Symbol())   // => "Symbol"
classof({})         // => "Object"
classof([])         // => "Array"
classof(/./)        // => "RegExp"
classof(()=>{})     // => "Function"
classof(new Map())  // => "Map"
classof(new Set())  // => "Set"
classof(new Date()) // => "Date"
```

Prior to ES6, this special behavior of the `Object.prototype.toString()` method was available only to instances of built-in types, and if you called this `classof()` function on an instance of a class you had defined yourself, it would simply return "Object". In ES6, however, `Object.prototype.toString()` looks for a property with the symbolic name `Symbol.toStringTag` on its argument, and if such a property exists, it uses the property value in its output. This means that if you define a class of your own, you can easily make it work with functions like `classof()`:

```
class Range {
    get [Symbol.toStringTag]() { return "Range"; }
    // the rest of this class is omitted here
}
let r = new Range(1, 10);
Object.prototype.toString.call(r)   // => "[object Range]"
classof(r)                          // => "Range"
```

14.4.4 Symbol.species

Prior to ES6, JavaScript did not provide any real way to create robust subclasses of built-in classes like Array. In ES6, however, you can extend any built-in class simply by using the `class` and `extends` keywords. §9.5.2 demonstrated that with this simple subclass of Array:

```
// A trivial Array subclass that adds getters for the first and last elements.
class EZArray extends Array {
    get first() { return this[0]; }
    get last() { return this[this.length-1]; }
}

let e = new EZArray(1,2,3);
let f = e.map(x => x * x);
e.last  // => 3: the last element of EZArray e
f.last  // => 9: f is also an EZArray with a last property
```

Array defines methods `concat()`, `filter()`, `map()`, `slice()`, and `splice()`, which return arrays. When we create an array subclass like EZArray that inherits these methods, should the inherited method return instances of Array or instances of EZArray? Good arguments can be made for either choice, but the ES6 specification says that (by default) the five array-returning methods will return instances of the subclass.

Here's how it works:

- In ES6 and later, the `Array()` constructor has a property with the symbolic name `Symbol.species`. (Note that this Symbol is used as the name of a property of the constructor function. Most of the other well-known Symbols described here are used as the name of methods of a prototype object.)
- When we create a subclass with `extends`, the resulting subclass constructor inherits properties from the superclass constructor. (This is in addition to the normal kind of inheritance, where instances of the subclass inherit methods of the superclass.) This means that the constructor for every subclass of Array also has an inherited property with name `Symbol.species`. (Or a subclass can define its own property with this name, if it wants.)
- Methods like `map()` and `slice()` that create and return new arrays are tweaked slightly in ES6 and later. Instead of just creating a regular Array, they (in effect) invoke `new this.constructor[Symbol.species]()` to create the new array.

Now here's the interesting part. Suppose that `Array[Symbol.species]` was just a regular data property, defined like this:

```
Array[Symbol.species] = Array;
```

In that case, then subclass constructors would inherit the `Array()` constructor as their "species," and invoking `map()` on an array subclass would return an instance of the superclass rather than an instance of the subclass. That is not how ES6 actually behaves, however. The reason is that `Array[Symbol.species]` is a read-only accessor property whose getter function simply returns `this`. Subclass constructors inherit this getter function, which means that by default, every subclass constructor is its own "species."

Sometimes this default behavior is not what you want, however. If you wanted the array-returning methods of EZArray to return regular Array objects, you just need to set `EZArray[Symbol.species]` to `Array`. But since the inherited property is a read-only accessor, you can't just set it with an assignment operator. You can use `defineProperty()`, however:

```
EZArray[Symbol.species] = Array; // Attempt to set a read-only property fails

// Instead we can use defineProperty():
Object.defineProperty(EZArray, Symbol.species, {value: Array});
```

The simplest option is probably to explicitly define your own `Symbol.species` getter when creating the subclass in the first place:

```
class EZArray extends Array {
    static get [Symbol.species]() { return Array; }
    get first() { return this[0]; }
    get last() { return this[this.length-1]; }
}

let e = new EZArray(1,2,3);
let f = e.map(x => x - 1);
e.last  // => 3
f.last  // => undefined: f is a regular array with no last getter
```

Creating useful subclasses of Array was the primary use case that motivated the introduction of `Symbol.species`, but it is not the only place that this well-known Symbol is used. Typed array classes use the Symbol in the same way that the Array class does. Similarly, the `slice()` method of ArrayBuffer looks at the `Symbol.species` property of `this.constructor` instead of simply creating a new ArrayBuffer. And Promise methods like `then()` that return new Promise objects create those objects via this species protocol as well. Finally, if you find yourself subclassing Map (for example) and defining methods that return new Map objects, you might want to use `Symbol.species` yourself for the benefit of subclasses of your subclass.

14.4.5 Symbol.isConcatSpreadable

The Array method `concat()` is one of the methods described in the previous section that uses `Symbol.species` to determine what constructor to use for the returned array. But `concat()` also uses `Symbol.isConcatSpreadable`. Recall from §7.8.3 that the `concat()` method of an array treats its `this` value and its array arguments differently than its nonarray arguments: nonarray arguments are simply appended to the new array, but the `this` array and any array arguments are flattened or "spread" so that the elements of the array are concatenated rather than the array argument itself.

Before ES6, `concat()` just used `Array.isArray()` to determine whether to treat a value as an array or not. In ES6, the algorithm is changed slightly: if the argument (or

the `this` value) to `concat()` is an object and has a property with the symbolic name `Symbol.isConcatSpreadable`, then the boolean value of that property is used to determine whether the argument should be "spread." If no such property exists, then `Array.isArray()` is used as in previous versions of the language.

There are two cases when you might want to use this Symbol:

- If you create an Array-like (see §7.9) object and want it to behave like a real array when passed to `concat()`, you can simply add the symbolic property to your object:

```
let arraylike = {
    length: 1,
    0: 1,
    [Symbol.isConcatSpreadable]: true
};
[].concat(arraylike)  // => [1]: (would be [[1]] if not spread)
```

- Array subclasses are spreadable by default, so if you are defining an array subclass that you do not want to act like an array when used with `concat()`, then you can[1] add a getter like this to your subclass:

```
class NonSpreadableArray extends Array {
    get [Symbol.isConcatSpreadable]() { return false; }
}
let a = new NonSpreadableArray(1,2,3);
[].concat(a).length // => 1; (would be 3 elements long if a was spread)
```

14.4.6 Pattern-Matching Symbols

§11.3.2 documented the String methods that perform pattern-matching operations using a RegExp argument. In ES6 and later, these methods have been generalized to work with RegExp objects or any object that defines pattern-matching behavior via properties with symbolic names. For each of the string methods `match()`, `matchAll()`, `search()`, `replace()`, and `split()`, there is a corresponding well-known Symbol: `Symbol.match`, `Symbol.search`, and so on.

RegExps are a general and very powerful way to describe textual patterns, but they can be complicated and not well suited to fuzzy matching. With the generalized string methods, you can define your own pattern classes using the well-known Symbol methods to provide custom matching. For example, you could perform string comparisons using Intl.Collator (see §11.7.3) to ignore accents when matching. Or you could define a pattern class based on the *Soundex* algorithm to match words based on

1 A bug in the V8 JavaScript engine means that this code does not work correctly in Node 13.

their approximate sounds or to loosely match strings up to a given Levenshtein distance.

In general, when you invoke one of these five String methods on a pattern object like this:

```
string.method(pattern, arg)
```

that invocation turns into an invocation of a symbolically named method on your pattern object:

```
pattern[symbol](string, arg)
```

As an example, consider the pattern-matching class in the next example, which implements pattern matching using the simple * and ? wildcards that you are probably familar with from filesystems. This style of pattern matching dates back to the very early days of the Unix operating system, and the patterns are often called *globs*:

```
class Glob {
    constructor(glob) {
        this.glob = glob;

        // We implement glob matching using RegExp internally.
        // ? matches any one character except /, and * matches zero or more
        // of those characters. We use capturing groups around each.
        let regexpText = glob.replace("?", "([^/])").replace("*", "([^/]*)");

        // We use the u flag to get Unicode-aware matching.
        // Globs are intended to match entire strings, so we use the ^ and $
        // anchors and do not implement search() or matchAll() since they
        // are not useful with patterns like this.
        this.regexp = new RegExp(`^${regexpText}$`, "u");
    }

    toString() { return this.glob; }

    [Symbol.search](s) { return s.search(this.regexp); }
    [Symbol.match](s)  { return s.match(this.regexp); }
    [Symbol.replace](s, replacement) {
        return s.replace(this.regexp, replacement);
    }
}

let pattern = new Glob("docs/*.txt");
"docs/js.txt".search(pattern)   // => 0: matches at character 0
"docs/js.htm".search(pattern)   // => -1: does not match
let match = "docs/js.txt".match(pattern);
match[0]     // => "docs/js.txt"
match[1]     // => "js"
match.index  // => 0
"docs/js.txt".replace(pattern, "web/$1.htm")  // => "web/js.htm"
```

14.4.7 Symbol.toPrimitive

§3.9.3 explained that JavaScript has three slightly different algorithms for converting objects to primitive values. Loosely speaking, for conversions where a string value is expected or preferred, JavaScript invokes an object's `toString()` method first and falls back on the `valueOf()` method if `toString()` is not defined or does not return a primitive value. For conversions where a numeric value is preferred, JavaScript tries the `valueOf()` method first and falls back on `toString()` if `valueOf()` is not defined or if it does not return a primitive value. And finally, in cases where there is no preference, it lets the class decide how to do the conversion. Date objects convert using `toString()` first, and all other types try `valueOf()` first.

In ES6, the well-known Symbol `Symbol.toPrimitive` allows you to override this default object-to-primitive behavior and gives you complete control over how instances of your own classes will be converted to primitive values. To do this, define a method with this symbolic name. The method must return a primitive value that somehow represents the object. The method you define will be invoked with a single string argument that tells you what kind of conversion JavaScript is trying to do on your object:

- If the argument is `"string"`, it means that JavaScript is doing the conversion in a context where it would expect or prefer (but not require) a string. This happens when you interpolate the object into a template literal, for example.
- If the argument is `"number"`, it means that JavaScript is doing the conversion in a context where it would expect or prefer (but not require) a numeric value. This happens when you use the object with a `<` or `>` operator or with arithmetic operators like `-` and `*`.
- If the argument is `"default"`, it means that JavaScript is converting your object in a context where either a numeric or string value could work. This happens with the `+`, `==`, and `!=` operators.

Many classes can ignore the argument and simply return the same primitive value in all cases. If you want instances of your class to be comparable and sortable with `<` and `>`, then that is a good reason to define a `[Symbol.toPrimitive]` method.

14.4.8 Symbol.unscopables

The final well-known Symbol that we'll cover here is an obscure one that was introduced as a workaround for compatibility issues caused by the deprecated `with` statement. Recall that the `with` statement takes an object and executes its statement body as if it were in a scope where the properties of that object were variables. This caused compatibility problems when new methods were added to the Array class, and it broke some existing code. `Symbol.unscopables` is the result. In ES6 and later, the

`with` statement has been slightly modified. When used with an object `o`, a `with` statement computes `Object.keys(o[Symbol.unscopables]||{})` and ignores properties whose names are in the resulting array when creating the simulated scope in which to execute its body. ES6 uses this to add new methods to `Array.prototype` without breaking existing code on the web. This means that you can find a list of the newest Array methods by evaluating:

```
let newArrayMethods = Object.keys(Array.prototype[Symbol.unscopables]);
```

14.5 Template Tags

Strings within backticks are known as "template literals" and were covered in §3.3.4. When an expression whose value is a function is followed by a template literal, it turns into a function invocation, and we call it a "tagged template literal." Defining a new tag function for use with tagged template literals can be thought of as metaprogramming, because tagged templates are often used to define DSLs—domain-specific languages—and defining a new tag function is like adding new syntax to JavaScript. Tagged template literals have been adopted by a number of frontend JavaScript packages. The GraphQL query language uses a ``gql`` `` tag function to allow queries to be embedded within JavaScript code. And the Emotion library uses a ``css`` `` tag function to enable CSS styles to be embedded in JavaScript. This section demonstrates how to write your own tag functions like these.

There is nothing special about tag functions: they are ordinary JavaScript functions, and no special syntax is required to define them. When a function expression is followed by a template literal, the function is invoked. The first argument is an array of strings, and this is followed by zero or more additional arguments, which can have values of any type.

The number of arguments depends on the number of values that are interpolated into the template literal. If the template literal is simply a constant string with no interpolations, then the tag function will be called with an array of that one string and no additional arguments. If the template literal includes one interpolated value, then the tag function is called with two arguments. The first is an array of two strings, and the second is the interpolated value. The strings in that initial array are the string to the left of the interpolated value and the string to its right, and either one of them may be the empty string. If the template literal includes two interpolated values, then the tag function is invoked with three arguments: an array of three strings and the two interpolated values. The three strings (any or all of which may be empty) are the text to the left of the first value, the text between the two values, and the text to the right of the second value. In the general case, if the template literal has `n` interpolated values, then the tag function will be invoked with `n+1` arguments. The first argument will be an array of `n+1` strings, and the remaining arguments are the `n` interpolated values, in the order that they appear in the template literal.

The value of a template literal is always a string. But the value of a tagged template literal is whatever value the tag function returns. This may be a string, but when the tag function is used to implement a DSL, the return value is typically a non-string data structure that is a parsed representation of the string.

As an example of a template tag function that returns a string, consider the following `html``` template, which is useful when you want to safely interpolate values into a string of HTML. The tag performs HTML escaping on each of the values before using it to build the final string:

```
function html(strings, ...values) {
    // Convert each value to a string and escape special HTML characters
    let escaped = values.map(v => String(v)
                                .replace("&", "&amp;")
                                .replace("<", "&lt;")
                                .replace(">", "&gt;")
                                .replace('"', "&quot;")
                                .replace("'", "&#39;"));

    // Return the concatenated strings and escaped values
    let result = strings[0];
    for(let i = 0; i < escaped.length; i++) {
        result += escaped[i] + strings[i+1];
    }
    return result;
}

let operator = "<";
html`<b>x ${operator} y</b>`              // => "<b>x &lt; y</b>"

let kind = "game", name = "D&D";
html`<div class="${kind}">${name}</div>` // =>'<div class="game">D&amp;D</div>'
```

For an example of a tag function that does not return a string but instead a parsed representation of a string, think back to the Glob pattern class defined in §14.4.6. Since the `Glob()` constructor takes a single string argument, we can define a tag function for creating new Glob objects:

```
function glob(strings, ...values) {
    // Assemble the strings and values into a single string
    let s = strings[0];
    for(let i = 0; i < values.length; i++) {
        s += values[i] + strings[i+1];
    }
    // Return a parsed representation of that string
    return new Glob(s);
}

let root = "/tmp";
let filePattern = glob`${root}/*.html`;  // A RegExp alternative
"/tmp/test.html".match(filePattern)[1]   // => "test"
```

One of the features mentioned in passing in §3.3.4 is the `String.raw``` tag function that returns a string in its "raw" form without interpreting any of the backslash escape sequences. This is implemented using a feature of tag function invocation that we have not discussed yet. When a tag function is invoked, we've seen that its first argument is an array of strings. But this array also has a property named `raw`, and the value of that property is another array of strings, with the same number of elements. The argument array includes strings that have had escape sequences interpreted as usual. And the raw array includes strings in which escape sequences are not interpreted. This obscure feature is important if you want to define a DSL with a grammar that uses backslashes. For example, if we wanted our `glob``` tag function to support pattern matching on Windows-style paths (which use backslashes instead of forward slashes) and we did not want users of the tag to have to double every backslash, we could rewrite that function to use `strings.raw[]` instead of `strings[]`. The downside, of course, would be that we could no longer use escapes like `\u` in our glob literals.

14.6 The Reflect API

The Reflect object is not a class; like the Math object, its properties simply define a collection of related functions. These functions, added in ES6, define an API for "reflecting upon" objects and their properties. There is little new functionality here: the Reflect object defines a convenient set of functions, all in a single namespace, that mimic the behavior of core language syntax and duplicate the features of various preexisting Object functions.

Although the `Reflect` functions do not provide any new features, they do group the features together in one convenient API. And, importantly, the set of `Reflect` functions maps one-to-one with the set of Proxy handler methods that we'll learn about in §14.7.

The Reflect API consists of the following functions:

`Reflect.apply(f, o, args)`
: This function invokes the function `f` as a method of `o` (or invokes it as a function with no `this` value if `o` is `null`) and passes the values in the `args` array as arguments. It is equivalent to `f.apply(o, args)`.

`Reflect.construct(c, args, newTarget)`
: This function invokes the constructor `c` as if the `new` keyword had been used and passes the elements of the array `args` as arguments. If the optional `newTarget` argument is specified, it is used as the value of `new.target` within the constructor invocation. If not specified, then the `new.target` value will be `c`.

`Reflect.defineProperty(o, name, descriptor)`
: This function defines a property on the object `o`, using `name` (a string or symbol) as the name of the property. The Descriptor object should define the value (or getter and/or setter) and attributes of the property. `Reflect.defineProperty()` is very similar to `Object.defineProperty()` but returns `true` on success and `false` on failures. (`Object.defineProperty()` returns `o` on success and throws TypeError on failure.)

`Reflect.deleteProperty(o, name)`
: This function deletes the property with the specified string or symbolic name from the object `o`, returning `true` if successful (or if no such property existed) and `false` if the property could not be deleted. Calling this function is similar to writing `delete o[name]`.

`Reflect.get(o, name, receiver)`
: This function returns the value of the property of `o` with the specified name (a string or symbol). If the property is an accessor method with a getter, and if the optional `receiver` argument is specified, then the getter function is called as a method of `receiver` instead of as a method of `o`. Calling this function is similar to evaluating `o[name]`.

`Reflect.getOwnPropertyDescriptor(o, name)`
: This function returns a property descriptor object that describes the attributes of the property named `name` of the object `o`, or returns `undefined` if no such property exists. This function is nearly identical to `Object.getOwnPropertyDescriptor()`, except that the Reflect API version of the function requires that the first argument be an object and throws TypeError if it is not.

`Reflect.getPrototypeOf(o)`
: This function returns the prototype of object `o` or `null` if the object has no prototype. It throws a TypeError if `o` is a primitive value instead of an object. This function is almost identical to `Object.getPrototypeOf()` except that `Object.getPrototypeOf()` only throws a TypeError for `null` and `undefined` arguments and coerces other primitive values to their wrapper objects.

`Reflect.has(o, name)`
: This function returns `true` if the object `o` has a property with the specified `name` (which must be a string or a symbol). Calling this function is similar to evaluating `name in o`.

`Reflect.isExtensible(o)`
: This function returns `true` if the object `o` is extensible (§14.2) and `false` if it is not. It throws a TypeError if `o` is not an object. `Object.isExtensible()` is similar but simply returns `false` when passed an argument that is not an object.

`Reflect.ownKeys(o)`
: This function returns an array of the names of the properties of the object `o` or throws a TypeError if `o` is not an object. The names in the returned array will be strings and/or symbols. Calling this function is similar to calling `Object.getOwnPropertyNames()` and `Object.getOwnPropertySymbols()` and combining their results.

`Reflect.preventExtensions(o)`
: This function sets the *extensible* attribute (§14.2) of the object `o` to `false` and returns `true` to indicate success. It throws a TypeError if `o` is not an object. `Object.preventExtensions()` has the same effect but returns `o` instead of `true` and does not throw TypeError for nonobject arguments.

`Reflect.set(o, name, value, receiver)`
: This function sets the property with the specified `name` of the object `o` to the specified `value`. It returns `true` on success and `false` on failure (which can happen if the property is read-only). It throws TypeError if `o` is not an object. If the specified property is an accessor property with a setter function, and if the optional `receiver` argument is passed, then the setter will be invoked as a method of `receiver` instead of being invoked as a method of `o`. Calling this function is usually the same as evaluating `o[name] = value`.

`Reflect.setPrototypeOf(o, p)`
: This function sets the prototype of the object `o` to `p`, returning `true` on success and `false` on failure (which can occur if `o` is not extensible or if the operation would cause a circular prototype chain). It throws a TypeError if `o` is not an object or if `p` is neither an object nor `null`. `Object.setPrototypeOf()` is similar, but returns `o` on success and throws TypeError on failure. Remember that calling either of these functions is likely to make your code slower by disrupting JavaScript interpreter optimizations.

14.7 Proxy Objects

The Proxy class, available in ES6 and later, is JavaScript's most powerful metaprogramming feature. It allows us to write code that alters the fundamental behavior of JavaScript objects. The Reflect API described in §14.6 is a set of functions that gives us direct access to a set of fundamental operations on JavaScript objects. What the Proxy class does is allows us a way to implement those fundamental operations ourselves and create objects that behave in ways that are not possible for ordinary objects.

When we create a Proxy object, we specify two other objects, the target object and the handlers object:

```
let proxy = new Proxy(target, handlers);
```

The resulting Proxy object has no state or behavior of its own. Whenever you perform an operation on it (read a property, write a property, define a new property, look up the prototype, invoke it as a function), it dispatches those operations to the handlers object or to the target object.

The operations supported by Proxy objects are the same as those defined by the Reflect API. Suppose that `p` is a Proxy object and you write `delete p.x`. The `Reflect.deleteProperty()` function has the same behavior as the `delete` operator. And when you use the `delete` operator to delete a property of a Proxy object, it looks for a `deleteProperty()` method on the handlers object. If such a method exists, it invokes it. And if no such method exists, then the Proxy object performs the property deletion on the target object instead.

Proxies work this way for all of the fundamental operations: if an appropriate method exists on the handlers object, it invokes that method to perform the operation. (The method names and signatures are the same as those of the Reflect functions covered in §14.6.) And if that method does not exist on the handlers object, then the Proxy performs the fundamental operation on the target object. This means that a Proxy can obtain its behavior from the target object or from the handlers object. If the handlers object is empty, then the proxy is essentially a transparent wrapper around the target object:

```
let t = { x: 1, y: 2 };
let p = new Proxy(t, {});
p.x          // => 1
delete p.y   // => true: delete property y of the proxy
t.y          // => undefined: this deletes it in the target, too
p.z = 3;     // Defining a new property on the proxy
t.z          // => 3: defines the property on the target
```

This kind of transparent wrapper proxy is essentially equivalent to the underlying target object, which means that there really isn't a reason to use it instead of the wrapped object. Transparent wrappers can be useful, however, when created as "revocable proxies." Instead of creating a Proxy with the `Proxy()` constructor, you can use the `Proxy.revocable()` factory function. This function returns an object that includes a Proxy object and also a `revoke()` function. Once you call the `revoke()` function, the proxy immediately stops working:

```
function accessTheDatabase() { /* implementation omitted */ return 42; }
let {proxy, revoke} = Proxy.revocable(accessTheDatabase, {});

proxy()   // => 42: The proxy gives access to the underlying target function
revoke(); // But that access can be turned off whenever we want
proxy();  // !TypeError: we can no longer call this function
```

Note that in addition to demonstrating revocable proxies, the preceding code also demonstrates that proxies can work with target functions as well as target objects. But the main point here is that revocable proxies are a building block for a kind of code isolation, and you might use them when dealing with untrusted third-party libraries, for example. If you have to pass a function to a library that you don't control, you can pass a revocable proxy instead and then revoke the proxy when you are finished with the library. This prevents the library from keeping a reference to your function and calling it at unexpected times. This kind of defensive programming is not typical in JavaScript programs, but the Proxy class at least makes it possible.

If we pass a non-empty handlers object to the `Proxy()` constructor, then we are no longer defining a transparent wrapper object and are instead implementing custom behavior for our proxy. With the right set of handlers, the underlying target object essentially becomes irrelevant.

In the following code, for example, is how we could implement an object that appears to have an infinite number of read-only properties, where the value of each property is the same as the name of the property:

```
// We use a Proxy to create an object that appears to have every
// possible property, with the value of each property equal to its name
let identity = new Proxy({}, {
    // Every property has its own name as its value
    get(o, name, target) { return name; },
    // Every property name is defined
    has(o, name) { return true; },
    // There are too many properties to enumerate, so we just throw
    ownKeys(o) { throw new RangeError("Infinite number of properties"); },
    // All properties exist and are not writable, configurable or enumerable.
    getOwnPropertyDescriptor(o, name) {
        return {
            value: name,
            enumerable: false,
            writable: false,
            configurable: false
        };
    },
    // All properties are read-only so they can't be set
    set(o, name, value, target) { return false; },
    // All properties are non-configurable, so they can't be deleted
    deleteProperty(o, name) { return false; },
    // All properties exist and are non-configurable so we can't define more
    defineProperty(o, name, desc) { return false; },
    // In effect, this means that the object is not extensible
    isExtensible(o) { return false; },
    // All properties are already defined on this object, so it couldn't
    // inherit anything even if it did have a prototype object.
    getPrototypeOf(o) { return null; },
    // The object is not extensible, so we can't change the prototype
    setPrototypeOf(o, proto) { return false; },
```

```
});

identity.x                // => "x"
identity.toString         // => "toString"
identity[0]               // => "0"
identity.x = 1;           // Setting properties has no effect
identity.x                // => "x"
delete identity.x         // => false: can't delete properties either
identity.x                // => "x"
Object.keys(identity);    // !RangeError: can't list all the keys
for(let p of identity) ;  // !RangeError
```

Proxy objects can derive their behavior from the target object and from the handlers object, and the examples we have seen so far have used one object or the other. But it is typically more useful to define proxies that use both objects.

The following code, for example, uses Proxy to create a read-only wrapper for a target object. When code tries to read values from the object, those reads are forwarded to the target object normally. But if any code tries to modify the object or its properties, methods of the handler object throw a TypeError. A proxy like this might be helpful for writing tests: suppose you've written a function that takes an object argument and want to ensure that your function does not make any attempt to modify the input argument. If your test passes in a read-only wrapper object, then any writes will throw exceptions that cause the test to fail:

```
function readOnlyProxy(o) {
    function readonly() { throw new TypeError("Readonly"); }
    return new Proxy(o, {
        set: readonly,
        defineProperty: readonly,
        deleteProperty: readonly,
        setPrototypeOf: readonly,
    });
}

let o = { x: 1, y: 2 };    // Normal writable object
let p = readOnlyProxy(o);  // Readonly version of it
p.x                        // => 1: reading properties works
p.x = 2;                   // !TypeError: can't change properties
delete p.y;                // !TypeError: can't delete properties
p.z = 3;                   // !TypeError: can't add properties
p.__proto__ = {};          // !TypeError: can't change the prototype
```

Another technique when writing proxies is to define handler methods that intercept operations on an object but still delegate the operations to the target object. The functions of the Reflect API (§14.6) have exactly the same signatures as the handler methods, so they make it easy to do that kind of delegation.

Here, for example, is a proxy that delegates all operations to the target object but uses handler methods to log the operations:

```
/*
 * Return a Proxy object that wraps o, delegating all operations to
 * that object after logging each operation. objname is a string that
 * will appear in the log messages to identify the object. If o has own
 * properties whose values are objects or functions, then if you query
 * the value of those properties, you'll get a loggingProxy back, so that
 * logging behavior of this proxy is "contagious".
 */
function loggingProxy(o, objname) {
    // Define handlers for our logging Proxy object.
    // Each handler logs a message and then delegates to the target object.
    const handlers = {
        // This handler is a special case because for own properties
        // whose value is an object or function, it returns a proxy rather
        // than returning the value itself.
        get(target, property, receiver) {
            // Log the get operation
            console.log(`Handler get(${objname},${property.toString()})`);

            // Use the Reflect API to get the property value
            let value = Reflect.get(target, property, receiver);

            // If the property is an own property of the target and
            // the value is an object or function then return a Proxy for it.
            if (Reflect.ownKeys(target).includes(property) &&
                (typeof value === "object" || typeof value === "function")) {
                return loggingProxy(value, `${objname}.${property.toString()}`);
            }

            // Otherwise return the value unmodified.
            return value;
        },

        // There is nothing special about the following three methods:
        // they log the operation and delegate to the target object.
        // They are a special case simply so we can avoid logging the
        // receiver object which can cause infinite recursion.
        set(target, prop, value, receiver) {
            console.log(`Handler set(${objname},${prop.toString()},${value})`);
            return Reflect.set(target, prop, value, receiver);
        },
        apply(target, receiver, args) {
            console.log(`Handler ${objname}(${args})`);
            return Reflect.apply(target, receiver, args);
        },
        construct(target, args, receiver) {
            console.log(`Handler ${objname}(${args})`);
            return Reflect.construct(target, args, receiver);
        }
    };

    // We can automatically generate the rest of the handlers.
```

```
    // Metaprogramming FTW!
    Reflect.ownKeys(Reflect).forEach(handlerName => {
        if (!(handlerName in handlers)) {
            handlers[handlerName] = function(target, ...args) {
                // Log the operation
                console.log(`Handler ${handlerName}(${objname},${args})`);
                // Delegate the operation
                return Reflect[handlerName](target, ...args);
            };
        }
    });

    // Return a proxy for the object using these logging handlers
    return new Proxy(o, handlers);
}
```

The `loggingProxy()` function defined earlier creates proxies that log all of the ways they are used. If you are trying to understand how an undocumented function uses the objects you pass it, using a logging proxy can help.

Consider the following examples, which result in some genuine insights about array iteration:

```
// Define an array of data and an object with a function property
let data = [10,20];
let methods = { square: x => x*x };

// Create logging proxies for the array and the object
let proxyData = loggingProxy(data, "data");
let proxyMethods = loggingProxy(methods, "methods");

// Suppose we want to understand how the Array.map() method works
data.map(methods.square)        // => [100, 400]

// First, let's try it with a logging Proxy array
proxyData.map(methods.square)   // => [100, 400]
// It produces this output:
// Handler get(data,map)
// Handler get(data,length)
// Handler get(data,constructor)
// Handler has(data,0)
// Handler get(data,0)
// Handler has(data,1)
// Handler get(data,1)

// Now lets try with a proxy methods object
data.map(proxyMethods.square)   // => [100, 400]
// Log output:
// Handler get(methods,square)
// Handler methods.square(10,0,10,20)
// Handler methods.square(20,1,10,20)
```

```
// Finally, let's use a logging proxy to learn about the iteration protocol
for(let x of proxyData) console.log("Datum", x);
// Log output:
// Handler get(data,Symbol(Symbol.iterator))
// Handler get(data,length)
// Handler get(data,0)
// Datum 10
// Handler get(data,length)
// Handler get(data,1)
// Datum 20
// Handler get(data,length)
```

From the first chunk of logging output, we learn that the `Array.map()` method explicitly checks for the existence of each array element (causing the `has()` handler to be invoked) before actually reading the element value (which triggers the `get()` handler). This is presumably so that it can distinguish nonexistent array elements from elements that exist but are undefined.

The second chunk of logging output might remind us that the function we pass to `Array.map()` is invoked with three arguments: the element's value, the element's index, and the array itself. (There is a problem in our logging output: the `Array.toString()` method does not include square brackets in its output, and the log messages would be clearer if they were included in the argument list `(10,0,[10,20])`.)

The third chunk of logging output shows us that the `for/of` loop works by looking for a method with symbolic name `[Symbol.iterator]`. It also demonstrates that the Array class's implementation of this iterator method is careful to check the array length at every iteration and does not assume that the array length remains constant during the iteration.

14.7.1 Proxy Invariants

The `readOnlyProxy()` function defined earlier creates Proxy objects that are effectively frozen: any attempt to alter a property value or property attribute or to add or remove properties will throw an exception. But as long as the target object is not frozen, we'll find that if we can query the proxy with `Reflect.isExtensible()` and `Reflect.getOwnPropertyDescriptor()`, and it will tell us that we should be able to set, add, and delete properties. So `readOnlyProxy()` creates objects in an inconsistent state. We could fix this by adding `isExtensible()` and `getOwnPropertyDescriptor()` handlers, or we can just live with this kind of minor inconsistency.

The Proxy handler API allows us to define objects with major inconsistencies, however, and in this case, the Proxy class itself will prevent us from creating Proxy objects that are inconsistent in a bad way. At the start of this section, we described proxies as objects with no behavior of their own because they simply forward all operations to

the handlers object and the target object. But this is not entirely true: after forwarding an operation, the Proxy class performs some sanity checks on the result to ensure important JavaScript invariants are not being violated. If it detects a violation, the proxy will throw a TypeError instead of letting the operation proceed.

As an example, if you create a proxy for a non-extensible object, the proxy will throw a TypeError if the `isExtensible()` handler ever returns `true`:

```
let target = Object.preventExtensions({});
let proxy = new Proxy(target, { isExtensible() { return true; }});
Reflect.isExtensible(proxy);  // !TypeError: invariant violation
```

Relatedly, proxy objects for non-extensible targets may not have a `getPrototypeOf()` handler that returns anything other than the real prototype object of the target. Also, if the target object has nonwritable, nonconfigurable properties, then the Proxy class will throw a TypeError if the `get()` handler returns anything other than the actual value:

```
let target = Object.freeze({x: 1});
let proxy = new Proxy(target, { get() { return 99; }});
proxy.x;         // !TypeError: value returned by get() doesn't match target
```

Proxy enforces a number of additional invariants, almost all of them having to do with non-extensible target objects and nonconfigurable properties on the target object.

14.8 Summary

In this chapter, you have learned:

- JavaScript objects have an *extensible* attribute and object properties have *writable*, *enumerable*, and *configurable* attributes, as well as a value and a getter and/or setter attribute. You can use these attributes to "lock down" your objects in various ways, including creating "sealed" and "frozen" objects.
- JavaScript defines functions that allow you to traverse the prototype chain of an object and even to change the prototype of an object (though doing this can make your code slower).
- The properties of the `Symbol` object have values that are "well-known Symbols," which you can use as property or method names for the objects and classes that you define. Doing so allows you to control how your object interacts with JavaScript language features and with the core library. For example, well-known Symbols allow you to make your classes iterable and control the string that is displayed when an instance is passed to `Object.prototype.toString()`. Prior to ES6, this kind of customization was available only to the native classes that were built in to an implementation.

- Tagged template literals are a function invocation syntax, and defining a new tag function is kind of like adding a new literal syntax to the language. Defining a tag function that parses its template string argument allows you to embed DSLs within JavaScript code. Tag functions also provide access to a raw, unescaped form of string literals where backslashes have no special meaning.
- The Proxy class and the related Reflect API allow low-level control over the fundamental behaviors of JavaScript objects. Proxy objects can be used as optionally revocable wrappers to improve code encapsulation, and they can also be used to implement nonstandard object behaviors (like some of the special case APIs defined by early web browsers).

CHAPTER 15

JavaScript in Web Browsers

The JavaScript language was created in 1994 with the express purpose of enabling dynamic behavior in the documents displayed by web browsers. The language has evolved significantly since then, and at the same time, the scope and capabilities of the web platform have grown explosively. Today, JavaScript programmers can think of the web as a full-featured platform for application development. Web browsers specialize in the display of formatted text and images, but, like native operating systems, browsers also provide other services, including graphics, video, audio, networking, storage, and threading. JavaScript is the language that enables web applications to use the services provided by the web platform, and this chapter demonstrates how you can use the most important of these services.

The chapter begins with the web platform's programming model, explaining how scripts are embedded within HTML pages (§15.1) and how JavaScript code is triggered asynchronously by events (§15.2). The sections that follow this introductory material document the core JavaScript APIs that enable your web applications to:

- Control document content (§15.3) and style (§15.4)
- Determine the on-screen position of document elements (§15.5)
- Create reusable user interface components (§15.6)
- Draw graphics (§15.7 and §15.8)
- Play and generate sounds (§15.9)
- Manage browser navigation and history (§15.10)
- Exchange data over the network (§15.11)
- Store data on the user's computer (§15.12)
- Perform concurrent computation with threads (§15.13)

Client-Side JavaScript

In this book, and on the web, you'll see the term "client-side JavaScript." The term is simply a synonym for JavaScript written to run in a web browser, and it stands in contrast to "server-side" code, which runs in web servers.

The two "sides" refer to the two ends of the network connection that separate the web server and the web browser, and software development for the web typically requires code to be written on both "sides." Client-side and server-side are also often called "frontend" and "backend."

Previous editions of this book attempted to comprehensively cover all JavaScript APIs defined by web browsers, and as a result, this book was too long a decade ago. The number and complexity of web APIs has continued to grow, and I no longer think it makes sense to attempt to cover them all in one book. As of the seventh edition, my goal is to cover the JavaScript language definitively and to provide an in-depth introduction to using the language with Node and with web browsers. This chapter cannot cover all the web APIs, but it introduces the most important ones in enough detail that you can start using them right away. And, having learned about the core APIs covered here, you should be able to pick up new APIs (like those summarized in §15.15) when and if you need them.

Node has a single implementation and a single authoritative source for documentation. Web APIs, by contrast, are defined by consensus among the major web browser vendors, and the authoritative documentation takes the form of a specification intended for the C++ programmers who implement the API, not for the JavaScript programmers who will use it. Fortunately, Mozilla's "MDN web docs" project is a reliable and comprehensive source[1] for web API documentation.

Legacy APIs

In the 25 years since JavaScript was first released, browser vendors have been adding features and APIs for programmers to use. Many of those APIs are now obsolete. They include:

- Proprietary APIs that were never standardized and/or never implemented by other browser vendors. Microsoft's Internet Explorer defined a lot of these APIs.

1 Previous editions of this book had an extensive reference section covering the JavaScript standard library and web APIs. It was removed in the seventh edition because MDN has made it obsolete: today, it is quicker to look something up on MDN than it is to flip through a book, and my former colleagues at MDN do a better job at keeping their online documentation up to date than this book ever could.

Some (like the `innerHTML` property) proved useful and were eventually standardized. Others (like the `attachEvent()` method) have been obsolete for years.

- Inefficient APIs (like the `document.write()` method) that have such a severe performance impact that their use is no longer considered acceptable.
- Outdated APIs that have long since been replaced by new APIs for achieving the same thing. An example is `document.bgColor`, which was defined to allow JavaScript to set the background color of a document. With the advent of CSS, `document.bgColor` became a quaint special case with no real purpose.
- Poorly designed APIs that have been replaced by better ones. In the early days of the web, standards committees defined the key Document Object Model API in a language-agnostic way so that the same API could be used in Java programs to work with XML documents on and in JavaScript programs to work with HTML documents. This resulted in an API that was not well suited to the JavaScript language and that had features that web programmers didn't particularly care about. It took decades to recover from those early design mistakes, but today's web browsers support a much-improved Document Object Model.

Browser vendors may need to support these legacy APIs for the foreseeable future in order to ensure backward compatibility, but there is no longer any need for this book to document them or for you to learn about them. The web platform has matured and stabilized, and if you are a seasoned web developer who remembers the fourth or fifth edition of this book, then you may have as much outdated knowledge to forget as you have new material to learn.

15.1 Web Programming Basics

This section explains how JavaScript programs for the web are structured, how they are loaded into a web browser, how they obtain input, how they produce output, and how they run asynchronously by responding to events.

15.1.1 JavaScript in HTML <script> Tags

Web browsers display HTML documents. If you want a web browser to execute JavaScript code, you must include (or reference) that code from an HTML document, and this is what the HTML `<script>` tag does.

JavaScript code can appear inline within an HTML file between `<script>` and `</script>` tags. Here, for example, is an HTML file that includes a script tag with JavaScript code that dynamically updates one element of the document to make it behave like a digital clock:

```
<!DOCTYPE html>                 <!-- This is an HTML5 file -->
<html>                          <!-- The root element -->
<head>                          <!-- Title, scripts & styles can go here -->
<title>Digital Clock</title>
<style>                         /* A CSS stylesheet for the clock */
#clock {                        /* Styles apply to element with id="clock" */
  font: bold 24px sans-serif;   /* Use a big bold font */
  background: #ddf;             /* on a light bluish-gray background. */
  padding: 15px;                /* Surround it with some space */
  border: solid black 2px;      /* and a solid black border */
  border-radius: 10px;          /* with rounded corners. */
}
</style>
</head>
<body>                    <!-- The body holds the content of the document. -->
<h1>Digital Clock</h1>    <!-- Display a title. -->
<span id="clock"></span>  <!-- We will insert the time into this element. -->
<script>
// Define a function to display the current time
function displayTime() {
    let clock = document.querySelector("#clock"); // Get element with id="clock"
    let now = new Date();                         // Get current time
    clock.textContent = now.toLocaleTimeString(); // Display time in the clock
}
displayTime()                    // Display the time right away
setInterval(displayTime, 1000);  // And then update it every second.
</script>
</body>
</html>
```

Although JavaScript code can be embedded directly within a `<script>` tag, it is more common to instead use the `src` attribute of the `<script>` tag to specify the URL (an absolute URL or a URL relative to the URL of the HTML file being displayed) of a file containing JavaScript code. If we took the JavaScript code out of this HTML file and stored it in its own *scripts/digital_clock.js* file, then the `<script>` tag might reference that file of code like this:

```
<script src="scripts/digital_clock.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 a `src` attribute 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: HTML does not support a `<script/>` tag.

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.

Modules

§10.3 documents JavaScript modules and covers their `import` and `export` directives. If you have written your JavaScript program using modules (and have not used a code-bundling tool to combine all your modules into a single nonmodular file of JavaScript), then you must load the top-level module of your program with a `<script>` tag that has a `type="module"` attribute. If you do this, then the module you specify will be loaded, and all of the modules it imports will be loaded, and (recursively) all of the modules they import will be loaded. See §10.3.5 for complete details.

Specifying script type

In the early days of the web, it was thought that browsers might some day implement languages other than JavaScript, and programmers added attributes like `language="javascript"` and `type="application/javascript"` to their `<script>` tags. This is completely unnecessary. JavaScript is the default (and only) language of the web. The `language` attribute is deprecated, and there are only two reasons to use a `type` attribute on a `<script>` tag:

- To specify that the script is a module
- To embed data into a web page without displaying it (see §15.3.4)

When scripts run: async and deferred

When JavaScript was first added to web browsers, there was no API for traversing and manipulating the structure and content of an already rendered 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 in the process of loading. It did this by

using the `document.write()` method to inject HTML text into the document at the location of the script.

The use of `document.write()` is no longer considered good style, but the fact that it is possible means that when the HTML parser encounters a `<script>` element, it must, by default, run the script just to be sure that it doesn't output any HTML before it can resume parsing and rendering the document. This can dramatically slow down parsing and rendering of the web page.

Fortunately, this default *synchronous* or *blocking* script execution mode is not the only option. The `<script>` tag can have `defer` and `async` attributes, which 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. Note that these attributes are only meaningful when used in conjunction with the `src` attribute:

```
<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()` to generate HTML output, and that the browser, therefore, 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 fully loaded and parsed and is ready to be manipulated. The `async` attribute causes the browser to run the script as soon as possible but does not block document parsing while the script is being downloaded. If a `<script>` tag has both attributes, the `async` attribute takes precedence.

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.

Scripts with the `type="module"` attribute are, by default, executed after the document has loaded, as if they had a `defer` attribute. You can override this default with the `async` attribute, which will cause the code to be executed as soon as the module and all of its dependencies have loaded.

A simple alternative to the `async` and `defer` attributes—especially for code that is included directly in the HTML—is to simply put your scripts at the end of the HTML file. That way, the script can run knowing that the document content before it has been parsed and is ready to be manipulated.

Loading scripts on demand

Sometimes, you may have JavaScript code that is not used when a document first loads and is only needed if the user takes some action like clicking on a button or opening a menu. If you are developing your code using modules, you can load a module on demand with `import()`, as described in §10.3.6.

If you are not using modules, you can load a file of JavaScript on demand simply by adding a <script> tag to your document when you want the script to load:

```
// Asynchronously load and execute a script from a specified URL
// Returns a Promise that resolves when the script has loaded.
function importScript(url) {
    return new Promise((resolve, reject) => {
        let s = document.createElement("script"); // Create a <script> element
        s.onload = () => { resolve(); };          // Resolve promise when loaded
        s.onerror = (e) => { reject(e); };        // Reject on failure
        s.src = url;                              // Set the script URL
        document.head.append(s);                  // Add <script> to document
    });
}
```

This `importScript()` function uses DOM APIs (§15.3) to create a new `<script>` tag and add it to the document `<head>`. And it uses event handlers (§15.2) to determine when the script has loaded successfully or when loading has failed.

15.1.2 The Document Object Model

One of the most important objects in client-side JavaScript programming is the Document object—which represents the HTML document that is displayed in a browser window or tab. The API for working with HTML documents is known as the Document Object Model, or DOM, and it is covered in detail in §15.3. But the DOM is so central to client-side JavaScript programming that it deserves to be introduced here.

HTML documents contain HTML elements nested within one another, forming a tree. Consider the following simple HTML document:

```
<html>
  <head>
    <title>Sample Document</title>
  </head>
  <body>
    <h1>An HTML Document</h1>
    <p>This is a <i>simple</i> document.
  </body>
</html>
```

The top-level `<html>` tag contains `<head>` and `<body>` tags. The `<head>` tag contains a `<title>` tag. And the `<body>` tag contains `<h1>` and `<p>` tags. The `<title>` and `<h1>` tags contain strings of text, and the `<p>` tag contains two strings of text with an `<i>` tag between them.

The DOM API mirrors the tree structure of an HTML document. For each HTML tag in the document, there is a corresponding JavaScript Element object, and for each run of text in the document, there is a corresponding Text object. The Element and

Text classes, as well as the Document class itself, are all subclasses of the more general Node class, and Node objects are organized into a tree structure that JavaScript can query and traverse using the DOM API. The DOM representation of this document is the tree pictured in Figure 15-1.

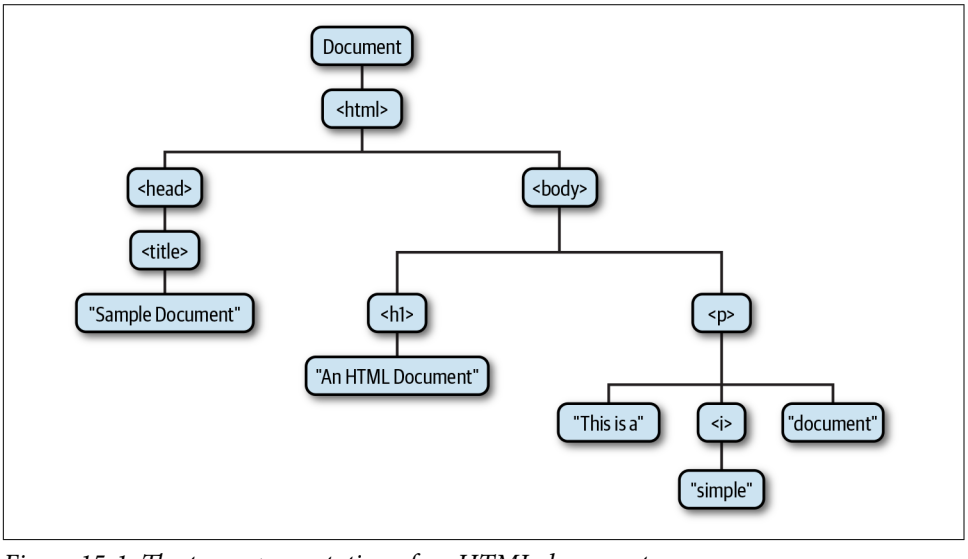

Figure 15-1. The tree representation of an HTML document

If you are not already familiar with tree structures in computer programming, it is helpful to know that they borrow terminology from family trees. The node directly above a node is the *parent* of that node. The nodes one level directly below another node are the *children* of that node. Nodes at the same level, and with the same parent, are *siblings*. The set of nodes any number of levels below another node are the *descendants* of that node. And the parent, grandparent, and all other nodes above a node are the *ancestors* of that node.

The DOM API includes methods for creating new Element and Text nodes, and for inserting them into the document as children of other Element objects. There are also methods for moving elements within the document and for removing them entirely. While a server-side application might produce plain-text output by writing strings with `console.log()`, a client-side JavaScript application can produce formatted HTML output by building or manipulating the document tree document using the DOM API.

There is a JavaScript class corresponding to each HTML tag type, and each occurrence of the tag in a document is represented by an instance of the class. The `<body>` tag, for example, is represented by an instance of HTMLBodyElement, and a `<table>` tag is represented by an instance of HTMLTableElement. The JavaScript element objects have properties that correspond to the HTML attributes of the tags. For

example, instances of HTMLImageElement, which represent `<img>` tags, have a `src` property that corresponds to the `src` attribute of the tag. The initial value of the `src` property is the attribute value that appears in the HTML tag, and setting this property with JavaScript changes the value of the HTML attribute (and causes the browser to load and display a new image). Most of the JavaScript element classes just mirror the attributes of an HTML tag, but some define additional methods. The HTMLAudioElement and HTMLVideoElement classes, for example, define methods like `play()` and `pause()` for controlling playback of audio and video files.

15.1.3 The Global Object in Web Browsers

There is one global object per browser window or tab (§3.7). All of the JavaScript code (except code running in worker threads; see §15.13) running in that window shares this single global object. This is true regardless of how many scripts or modules are in the document: all the scripts and modules of a document share a single global object; if one script defines a property on that object, that property is visible to all the other scripts as well.

The global object is where JavaScript's standard library is defined—the `parseInt()` function, the Math object, the Set class, and so on. In web browsers, the global object also contains the main entry points of various web APIs. For example, the `document` property represents the currently displayed document, the `fetch()` method makes HTTP network requests, and the `Audio()` constructor allows JavaScript programs to play sounds.

In web browsers, the global object does double duty: in addition to defining built-in types and functions, it also represents the current web browser window and defines properties like `history` (§15.10.2), which represent the window's browsing history, and `innerWidth`, which holds the window's width in pixels. One of the properties of this global object is named `window`, and its value is the global object itself. This means that you can simply type `window` to refer to the global object in your client-side code. When using window-specific features, it is often a good idea to include a `window.` prefix: `window.innerWidth` is clearer than `innerWidth`, for example.

15.1.4 Scripts Share a Namespace

With modules, the constants, variables, functions, and classes defined at the top level (i.e., outside of any function or class definition) of the module are private to the module unless they are explicitly exported, in which case, they can be selectively imported by other modules. (Note that this property of modules is honored by code-bundling tools as well.)

With non-module scripts, however, the situation is completely different. If the top-level code in a script defines a constant, variable, function, or class, that declaration

will be visible to all other scripts in the same document. If one script defines a function `f()` and another script defines a class `c`, then a third script can invoke the function and instantiate the class without having to take any action to import them. So if you are not using modules, the independent scripts in your document share a single namespace and behave as if they are all part of a single larger script. This can be convenient for small programs, but the need to avoid naming conflicts can become problematic for larger programs, especially when some of the scripts are third-party libraries.

There are some historical quirks with how this shared namespace works. `var` and `function` declarations at the top level create properties in the shared global object. If one script defines a top-level function `f()`, then another script in the same document can invoke that function as `f()` or as `window.f()`. On the other hand, the ES6 declarations `const`, `let`, and `class`, when used at the top level, do not create properties in the global object. They are still defined in a shared namespace, however: if one script defines a class `C`, other scripts will be able to create instances of that class with `new C()`, but not with `new window.C()`.

To summarize: in modules, top-level declarations are scoped to the module and can be explicitly exported. In nonmodule scripts, however, top-level declarations are scoped to the containing document, and the declarations are shared by all scripts in the document. Older `var` and `function` declarations are shared via properties of the global object. Newer `const`, `let`, and `class` declarations are also shared and have the same document scope, but they do not exist as properties of any object that JavaScript code has access to.

15.1.5 Execution of JavaScript Programs

There is no formal definition of a *program* in client-side JavaScript, but we can say that a JavaScript program consists of all the JavaScript code in, or referenced from, a document. These separate bits of code share a single global Window object, which gives them access to the same underlying Document object representing the HTML document. Scripts that are not modules additionally share a top-level namespace.

If a web page includes an embedded frame (using the `<iframe>` element), the JavaScript code in the embedded document has a different global object and Document 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 both loaded 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. §15.13.6 explains how a JavaScript program can send and receive messages to and from JavaScript code running in an `<iframe>`.

You can think of JavaScript program execution as occurring 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 run in the order in which they appear in the document, though this default order can be modified by the `async` and `defer` attributes we've described. The JavaScript code within any single script is run from top to bottom, subject, of course, to JavaScript's conditionals, loops, and other control statements. Some scripts don't really *do* anything during this first phase and instead just define functions and classes for use in the second phase. Other scripts might do significant work during the first phase and then do nothing in the second. Imagine a script at the very end of a document that finds all `<h1>` and `<h2>` tags in the document and modifies the document by generating and inserting a table of contents at the beginning of the document. This could be done entirely in the first phase. (See §15.3.6 for an example that does exactly this.)

Once the document is loaded and all scripts have run, JavaScript execution enters its second phase. This phase is asynchronous and event-driven. If a script is going to participate in this second phase, then one of the things it must have done during the first phase is to register at least one event handler or other callback function that will be invoked asynchronously. During this event-driven second phase, the web browser invokes event handler functions and other callbacks 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, document and resource loading, elapsed time, or errors in JavaScript code. Events and event handlers are described in detail in §15.2.

Some of the first events to occur during the event-driven phase are the "DOMContentLoaded" and "load" events. "DOMContentLoaded" is triggered when the HTML document has been completely loaded and parsed. The "load" event is triggered when all of the document's external resources—such as images—are also fully loaded. JavaScript programs often use one of these events as a trigger or starting signal. It is common to see programs whose scripts define functions but take no action other than registering an event handler function to be triggered by the "load" event at the beginning of the event-driven phase of execution. It is this "load" event handler that then manipulates the document and does whatever it is that the program is supposed to do. Note that it is common in JavaScript programming for an event handler function such as the "load" event handler described here to register other event handlers.

The loading phase of a JavaScript program is relatively short: ideally less than a second. 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. We'll cover these two phases in more detail next.

Client-side JavaScript threading model

JavaScript is a single-threaded language, and single-threaded execution makes for much simpler programming: 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 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.

The web platform 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 message 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 §15.13 for full details on the web's safe threading mechanism.

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. These two phases can be further broken down into the following steps:

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 a `<script>` tag that does not have any of the `async`, `defer`, or `type="module"` attributes, it adds that script tag to the document and then executes the script. The script is executed synchronously, and the HTML parser pauses while the script downloads (if necessary) and runs. A script like this can use `document.write()` to insert text into the input stream, and that text will become part of the document when the parser resumes. A script like this

often simply defines functions and registers event handlers for later use, but it can traverse and manipulate the document tree as it exists at that time. That is, non-module scripts that do not have an `async` or `defer` attribute can see their own `<script>` tag 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 if the script is a module, it also recursively downloads all of the script's dependencies) 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 `document.write()` method. They can see their own `<script>` tag and all document content that comes 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 (along with any module scripts that do not have an `async` attribute) 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 and they 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.

15.1.6 Program Input and Output

Like any program, client-side JavaScript programs process input data to produce output data. There are a variety of inputs available:

- The content of the document itself, which JavaScript code can access with the DOM API (§15.3).
- User input, in the form of events, such as mouse clicks (or touch-screen taps) on HTML `<button>` elements, or text entered into HTML `<textarea>` elements, for example. §15.2 demonstrates how JavaScript programs can respond to user events like these.
- The URL of the document being displayed is available to client-side JavaScript as `document.URL`. If you pass this string to the `URL()` constructor (§11.9), you can easily access the path, query, and fragment sections of the URL.
- The content of the HTTP "Cookie" request header is available to client-side code as `document.cookie`. Cookies are usually used by server-side code for maintaining user sessions, but client-side code can also read (and write) them if necessary. See §15.12.2 for further details.
- The global `navigator` property provides access to information about the web browser, the OS it's running on top of, and the capabilities of each. For example, `navigator.userAgent` is a string that identifies the web browser, `navigator.language` is the user's preferred language, and `navigator.hardwareConcurrency` returns the number of logical CPUs available to the web browser. Similarly, the global `screen` property provides access to the user's display size via the `screen.width` and `screen.height` properties. In a sense, these `navigator` and `screen` objects are to web browsers what environment variables are to Node programs.

Client-side JavaScript typically produces output, when it needs to, by manipulating the HTML document with the DOM API (§15.3) or by using a higher-level framework such as React or Angular to manipulate the document. Client-side code can also use `console.log()` and related methods (§11.8) to produce output. But this output is only visible in the web developer console, so it is useful when debugging, but not for user-visible output.

15.1.7 Program Errors

Unlike applications (such as Node applications) that run directly on top of the OS, JavaScript programs in a web browser can't really "crash." If an exception occurs while your JavaScript program is running, and if you do not have a `catch` statement to handle it, an error message will be displayed in the developer console, but any event handlers that have been registered keep running and responding to events.

If you would like to define an error handler of last resort to be invoked when this kind of uncaught exception occurs, set the `onerror` property of the Window object to an error handler function. When an uncaught exception propagates all the way up the

call stack and an error message is about to be displayed in the developer console, the `window.onerror` function will be invoked with three string arguments. The first argument to `window.onerror` is a message describing the error. The second argument is a string that contains the URL of the JavaScript code that caused the error. The third argument is the line number within the document where the error occurred. If the `onerror` handler returns `true`, it tells the browser that the handler has handled the error and that no further action is necessary—in other words, the browser should not display its own error message.

When a Promise is rejected and there is no `.catch()` function to handle it, that is a situation much like an unhandled exception: an unanticipated error or a logic error in your program. You can detect this by defining a `window.onunhandledrejection` function or by using `window.addEventListener()` to register a handler for "unhandledrejection" events. The event object passed to this handler will have a `promise` property whose value is the Promise object that rejected and a `reason` property whose value is what would have been passed to a `.catch()` function. As with the error handlers described earlier, if you call `preventDefault()` on the unhandled rejection event object, it will be considered handled and won't cause an error message in the developer console.

It is not often necessary to define `onerror` or `onunhandledrejection` handlers, but it can be quite useful as a telemetry mechanism if you want to report client-side errors to the server (using the `fetch()` function to make an HTTP POST request, for example) so that you can get information about unexpected errors that happen in your users' browsers.

15.1.8 The Web Security Model

The fact that web pages can execute arbitrary JavaScript code on your personal device 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

The subsections that follow give a quick overview of the security restrictions and issues that you, as a JavaScript programmer, should to be aware of.

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.

Similarly, client-side JavaScript does not have general-purpose networking capabilities. A client-side JavaScript program can make HTTP requests (§15.11.1). And another standard, known as WebSockets (§15.11.3), 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.

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. In this case, the same-origin policy governs the interactions of JavaScript code in one frame with the content of other 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.

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. Browsers typically treat every `file:` URL as a separate origin, which means that if you're working on a program that displays more than one document from the same server, you may not be able to test it locally using `file:` URLs and will have to run a static web server during development.

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 document contains an `<iframe>` that contains a second document from host B, then the script also has full access to the content of that second document. But if the top-level document contains another `<iframe>` that displays a document from host C (or even one from host A), then the same-origin policy comes into effect and prevents the script from accessing this nested document.

The same-origin policy also applies to scripted HTTP requests (see §15.11.1). JavaScript code can 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 (unless those web servers opt in with CORS, as we describe next).

The same-origin policy poses problems for large websites that use multiple subdomains. For example, scripts with origin *orders.example.com* might need to read properties from documents on *example.com*. To support multidomain websites of this sort, scripts can alter their origin by setting `document.domain` to a domain suffix. So a script with origin *https://orders.example.com* can change its origin to *https://example.com* by setting `document.domain` to "example.com." But that script cannot set `document.domain` to "orders.example", "ample.com", or "com".

The second technique for relaxing the same-origin policy is Cross-Origin Resource Sharing, or CORS, which allows servers to decide which origins they are willing to serve. CORS 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 honor these CORS headers and do not relax same-origin restrictions unless they are present.

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. Client-side JavaScript programmers must 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>
let name = new URL(document.URL).searchParams.get("name");
document.querySelector('h1').innerHTML = "Hello " + name;
</script>
```

This two-line script extracts input from the "name" query parameter of the document URL. It then uses the DOM API to inject an HTML string into the first `<h1>` tag in the document. This page is intended to be invoked with a URL like this:

```
http://www.example.com/greet.html?name=David
```

When used like this, it displays the text "Hello David." But consider what happens when it is invoked with this query parameter:

```
name=%3Cimg%20src=%22x.png%22%20onload=%22alert(%27hacked%27)%22/%3E
```

When the URL-escaped parameters are decoded, this URL causes the following HTML to be injected into the document:

```
Hello <img src="x.png" onload="alert('hacked')"/>
```

After the image loads, the string of JavaScript in the `onload` attribute is executed. The global `alert()` function displays a modal dialogue box. A single dialogue box is relatively benign but demonstrates that arbitrary code execution is possible on this site because it displays unsanitized HTML.

Cross-site scripting attacks are so called because more than one site is involved. Site B includes a specially crafted link (like the one in the previous example) to site A. If site B can convince users to click the link, they will be taken to site A, but that site will now be running code from site B. That code might deface the page or cause it to malfunction. More dangerously, the malicious code could read cookies stored by site A (perhaps account numbers or other personally identifying information) and send that data back to site B. The injected code could 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 replacing special HTML characters in the untrusted input string with their equivalent HTML entities:

```
name = name
    .replace(/&/g, "&amp;")
    .replace(/</g, "&lt;")
    .replace(/>/g, "&gt;")
    .replace(/"/g, "&quot;")
    .replace(/'/g, "&#x27;")
    .replace(/\//g, "&#x2F;")
```

Another approach to the problem of XSS is to structure your web applications so that untrusted content is always displayed in an `<iframe>` with the `sandbox` attribute set to disable scripting and other capabilities.

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.

15.2 Events

Client-side JavaScript programs use an asynchronous event-driven programming model. In this style of programming, the web browser generates an *event* whenever something interesting happens to the document or browser or to some element or object associated with it. For example, the web browser generates an event when it

finishes loading a document, when the user moves the mouse over a hyperlink, or when the user strikes a key on the keyboard. If a JavaScript application cares about a particular type of event, it can register one or more functions to be invoked when events of that type occur. Note that this is not unique to web programming: all applications with graphical user interfaces are designed this way—they sit around waiting to be interacted with (i.e., they wait for events to occur), and then they respond.

In client-side JavaScript, events can occur on any element within an HTML document, and this fact makes the event model of web browsers significantly more complex than Node's event model. We begin this section with some important definitions that help to explain that event model:

event type
: This string specifies what kind of event occurred. The type "mousemove," for example, means that the user moved the mouse. The type "keydown" means that the user pressed a key on the keyboard down. And the type "load" means that a document (or some other resource) has finished loading from the network. Because the type of an event is just a string, it's sometimes called an *event name*, and indeed, we use this name to identify the kind of event we're talking about.

event target
: This is the object on which the event occurred or with which the event is associated. When we speak of an event, we must specify both the type and the target. A load event on a Window, for example, or a click event on a `<button>` Element. Window, Document, and Element objects are the most common event targets in client-side JavaScript applications, but some events are triggered on other kinds of objects. For example, a Worker object (a kind of thread, covered §15.13) is a target for "message" events that occur when the worker thread sends a message to the main thread.

event handler, or event listener
: This function handles or responds to an event.[2] Applications register their event handler functions with the web browser, specifying an event type and an event target. When an event of the specified type occurs on the specified target, the browser invokes the handler function. When event handlers are invoked for an object, we say that the browser has "fired," "triggered," or "dispatched" the event. There are a number of ways to register event handlers, and the details of handler registration and invocation are explained in §15.2.2 and §15.2.3.

2 Some sources, including the HTML specification, make a technical distinction between handlers and listeners, based on the way in which they are registered. In this book, we treat the two terms as synonyms.

event object
: This object is associated with a particular event and contains details about that event. Event objects are passed as an argument to the event handler function. All event objects have a `type` property that specifies the event type and a `target` property that specifies the event target. Each event type defines a set of properties for its associated event object. The object associated with a mouse event includes the coordinates of the mouse pointer, for example, and the object associated with a keyboard event contains details about the key that was pressed and the modifier keys that were held down. Many event types define only a few standard properties—such as `type` and `target`—and do not carry much other useful information. For those events, it is the simple occurrence of the event, not the event details, that matter.

event propagation
: This is the process by which the browser decides which objects to trigger event handlers on. For events that are specific to a single object—such as the "load" event on the Window object or a "message" event on a Worker object—no propagation is required. But when certain kinds of events occur on elements within the HTML document, however, they propagate or "bubble" up the document tree. If the user moves the mouse over a hyperlink, the mousemove event is first fired on the `<a>` element that defines that link. Then it is fired on the containing elements: perhaps a `<p>` element, a `<section>` element, and the Document object itself. It is sometimes more convenient to register a single event handler on a Document or other container element than to register handlers on each individual element you're interested in. An event handler can stop the propagation of an event so that it will not continue to bubble and will not trigger handlers on containing elements. Handlers do this by invoking a method of the event object. In another form of event propagation, known as *event capturing*, handlers specially registered on container elements have the opportunity to intercept (or "capture") events before they are delivered to their actual target. Event bubbling and capturing are covered in detail in §15.2.4.

Some events have *default actions* associated with them. When a click event occurs on a hyperlink, for example, the default action is for the browser to follow the link and load a new page. Event handlers can prevent this default action by invoking a method of the event object. This is sometimes called "canceling" the event and is covered in §15.2.5.

15.2.1 Event Categories

Client-side JavaScript supports such a large number of event types that there is no way this chapter can cover them all. It can be useful, though, to group events into some general categories, to illustrate the scope and wide variety of supported events:

Device-dependent input events
: These events are directly tied to a specific input device, such as the mouse or keyboard. They include event types such as "mousedown," "mousemove," "mouseup," "touchstart," "touchmove," "touchend," "keydown," and "keyup."

Device-independent input events
: These input events are not directly tied to a specific input device. The "click" event, for example, indicates that a link or button (or other document element) has been activated. This is often done via a mouse click, but it could also be done by keyboard or (on touch-sensitive devices) with a tap. The "input" event is a device-independent alternative to the "keydown" event and supports keyboard input as well as alternatives such as cut-and-paste and input methods used for ideographic scripts. The "pointerdown," "pointermove," and "pointerup" event types are device-independent alternatives to mouse and touch events. They work for mouse-type pointers, for touch screens, and for pen- or stylus-style input as well.

User interface events
: UI events are higher-level events, often on HTML form elements that define a user interface for a web application. They include the "focus" event (when a text input field gains keyboard focus), the "change" event (when the user changes the value displayed by a form element), and the "submit" event (when the user clicks a Submit button in a form).

State-change events
: Some events are not triggered directly by user activity, but by network or browser activity, and indicate some kind of life-cycle or state-related change. The "load" and "DOMContentLoaded" events—fired on the Window and Document objects, respectively, at the end of document loading—are probably the most commonly used of these events (see "Client-side JavaScript timeline" on page 420). Browsers fire "online" and "offline" events on the Window object when network connectivity changes. The browser's history management mechanism (§15.10.4) fires the "popstate" event in response to the browser's Back button.

API-specific events
: A number of web APIs defined by HTML and related specifications include their own event types. The HTML `<video>` and `<audio>` elements define a long list of associated event types such as "waiting," "playing," "seeking," "volumechange," and so on, and you can use them to customize media playback. Generally

speaking, web platform APIs that are asynchronous and were developed before Promises were added to JavaScript are event-based and define API-specific events. The IndexedDB API, for example (§15.12.3), fires "success" and "error" events when database requests succeed or fail. And although the new `fetch()` API (§15.11.1) for making HTTP requests is Promise-based, the XMLHttpRequest API that it replaces defines a number of API-specific event types.

15.2.2 Registering Event Handlers

There are two basic ways to register event handlers. The first, from the early days of the web, is to set a property on the object or document element that is the event target. The second (newer and more general) technique is to pass the handler to the `addEventListener()` method of the object or element.

Setting event handler properties

The simplest way to register an event handler is by setting a property of the event target to the desired event handler function. By convention, event handler properties have names that consist of the word "on" followed by the event name: `onclick`, `onchange`, `onload`, `onmouseover`, and so on. Note that these property names are case sensitive and are written in all lowercase,[3] even when the event type (such as "mousedown") consists of multiple words. The following code includes two event handler registrations of this kind:

```
// Set the onload property of the Window object to a function.
// The function is the event handler: it is invoked when the document loads.
window.onload = function() {
    // Look up a <form> element
    let form = document.querySelector("form#shipping");
    // Register an event handler function on the form that will be invoked
    // before the form is submitted. Assume isFormValid() is defined elsewhere.
    form.onsubmit = function(event) { // When the user submits the form
        if (!isFormValid(this)) {     // check whether form inputs are valid
            event.preventDefault();   // and if not, prevent form submission.
        }
    };
};
```

The shortcoming of event handler properties is that they are designed around the assumption that event targets will have at most one handler for each type of event. It

3 If you have used the React framework to create client-side user interfaces, this may surprise you. React makes a number of minor changes to the client-side event model, and one of them is that in React, event handler property names are written in camelCase: `onClick`, `onMouseOver`, and so on. When working with the web platform natively, however, the event handler properties are written entirely in lowercase.

is often better to register event handlers using `addEventListener()` because that technique does not overwrite any previously registered handlers.

Setting event handler attributes

The event handler properties of document elements can also be defined directly in the HTML file as attributes on the corresponding HTML tag. (Handlers that would be registered on the Window element with JavaScript can be defined with attributes on the `<body>` tag in HTML.) This technique is generally frowned upon in modern web development, but it is possible, and it's documented here because you may still see it in existing code.

When defining an event handler as an HTML attribute, the attribute value should be a string of JavaScript code. That code should be the *body* of the event handler function, not a complete function declaration. That is, your HTML event handler code should not be surrounded by curly braces and prefixed with the `function` keyword. For example:

```
<button onclick="console.log('Thank you');">Please Click</button>
```

If an HTML event handler attribute contains multiple JavaScript statements, you must remember to separate those statements with semicolons or break the attribute value across multiple lines.

When you specify a string of JavaScript code as the value of an HTML event handler attribute, the browser converts your string into a function that works something like this one:

```
function(event) {
    with(document) {
        with(this.form || {}) {
            with(this) {
                /* your code here */
            }
        }
    }
}
```

The `event` argument means that your handler code can refer to the current event object as `event`. The `with` statements mean that the code of your handler can refer to the properties of the target object, the containing `<form>` (if any), and the containing Document object directly, as if they were variables in scope. The `with` statement is forbidden in strict mode (§5.6.3), but JavaScript code in HTML attributes is never strict. Event handlers defined in this way are executed in an environment in which unexpected variables are defined. This can be a source of confusing bugs and is a good reason to avoid writing event handlers in HTML.

addEventListener()

Any object that can be an event target—this includes the Window and Document objects and all document Elements—defines a method named `addEventListener()` that you can use to register an event handler for that target. `addEventListener()` takes three arguments. The first is the event type for which the handler is being registered. The event type (or name) is a string that does not include the "on" prefix used when setting event handler properties. The second argument to `addEventListener()` is the function that should be invoked when the specified type of event occurs. The third argument is optional and is explained below.

The following code registers two handlers for the "click" event on a `<button>` element. Note the differences between the two techniques used:

```
<button id="mybutton">Click me</button>
<script>
let b = document.querySelector("#mybutton");
b.onclick = function() { console.log("Thanks for clicking me!"); };
b.addEventListener("click", () => { console.log("Thanks again!"); });
</script>
```

Calling `addEventListener()` with "click" as its first argument does not affect the value of the `onclick` property. In this code, a button click will log two messages to the developer console. And if we called `addEventListener()` first and then set `onclick`, we would still log two messages, just in the opposite order. More importantly, you can call `addEventListener()` multiple times to register more than one handler function for the same event type on the same object. When an event occurs on an object, all of the handlers registered for that type of event are invoked in the order in which they were registered. Invoking `addEventListener()` more than once on the same object with the same arguments has no effect—the handler function remains registered only once, and the repeated invocation does not alter the order in which handlers are invoked.

`addEventListener()` is paired with a `removeEventListener()` method that expects the same two arguments (plus an optional third) but removes an event handler function from an object rather than adding it. It is often useful to temporarily register an event handler and then remove it soon afterward. For example, when you get a "mousedown" event, you might register temporary event handlers for "mousemove" and "mouseup" events so that you can see if the user drags the mouse. You'd then deregister these handlers when the "mouseup" event arrives. In such a situation, your event handler removal code might look like this:

```
document.removeEventListener("mousemove", handleMouseMove);
document.removeEventListener("mouseup", handleMouseUp);
```

The optional third argument to `addEventListener()` is a boolean value or object. If you pass `true`, then your handler function is registered as a *capturing* event handler

and is invoked at a different phase of event dispatch. We'll cover event capturing in §15.2.4. If you pass a third argument of `true` when you register an event listener, then you must also pass `true` as the third argument to `removeEventListener()` if you want to remove the handler.

Registering a capturing event handler is only one of the three options that `addEventListener()` supports, and instead of passing a single boolean value, you can also pass an object that explicitly specifies the options you want:

```
document.addEventListener("click", handleClick, {
    capture: true,
    once: true,
    passive: true
});
```

If the Options object has a `capture` property set to `true`, then the event handler will be registered as a capturing handler. If that property is `false` or is omitted, then the handler will be non-capturing.

If the Options object has a `once` property set to `true`, then the event listener will be automatically removed after it is triggered once. If this property is `false` or is omitted, then the handler is never automatically removed.

If the Options object has a `passive` property set to `true`, it indicates that the event handler will never call `preventDefault()` to cancel the default action (see §15.2.5). This is particularly important for touch events on mobile devices—if event handlers for "touchmove" events can prevent the browser's default scrolling action, then the browser cannot implement smooth scrolling. This `passive` property provides a way to register a potentially disruptive event handler of this sort but lets the web browser know that it can safely begin its default behavior—such as scrolling—while the event handler is running. Smooth scrolling is so important for a good user experience that Firefox and Chrome make "touchmove" and "mousewheel" events passive by default. So if you actually want to register a handler that calls `preventDefault()` for one of these events, you should explicitly set the `passive` property to `false`.

You can also pass an Options object to `removeEventListener()`, but the `capture` property is the only one that is relevant. There is no need to specify `once` or `passive` when removing a listener, and these properties are ignored.

15.2.3 Event Handler Invocation

Once you've registered an event handler, the web browser will invoke it automatically when an event of the specified type occurs on the specified object. This section describes event handler invocation in detail, explaining event handler arguments, the invocation context (the `this` value), and the meaning of the return value of an event handler.

Event handler argument

Event handlers are invoked with an Event object as their single argument. The properties of the Event object provide details about the event:

`type`
: The type of the event that occurred.

`target`
: The object on which the event occurred.

`currentTarget`
: For events that propagate, this property is the object on which the current event handler was registered.

`timeStamp`
: A timestamp (in milliseconds) that represents when the event occurred but that does not represent an absolute time. You can determine the elapsed time between two events by subtracting the timestamp of the first event from the timestamp of the second.

`isTrusted`
: This property will be `true` if the event was dispatched by the web browser itself and `false` if the event was dispatched by JavaScript code.

Specific kinds of events have additional properties. Mouse and pointer events, for example, have `clientX` and `clientY` properties that specify the window coordinates at which the event occurred.

Event handler context

When you register an event handler by setting a property, it looks as if you are defining a new method on the target object:

```
target.onclick = function() { /* handler code */ };
```

It isn't surprising, therefore, that event handlers are invoked as methods of the object on which they are defined. That is, within the body of an event handler, the `this` keyword refers to the object on which the event handler was registered.

Handlers are invoked with the target as their `this` value, even when registered using `addEventListener()`. This does not work for handlers defined as arrow functions, however: arrow functions always have the same `this` value as the scope in which they are defined.

Handler return value

In modern JavaScript, event handlers should not return anything. You may see event handlers that return values in older code, and the return value is typically a signal to the browser that it should not perform the default action associated with the event. If the `onclick` handler of a Submit button in a form returns `false`, for example, then the web browser will not submit the form (usually because the event handler determined that the user's input fails client-side validation).

The standard and preferred way to prevent the browser from performing a default action is to call the `preventDefault()` method (§15.2.5) on the Event object.

Invocation order

An event target may have more than one event handler registered for a particular type of event. When an event of that type occurs, the browser invokes all of the handlers in the order in which they were registered. Interestingly, this is true even if you mix event handlers registered with `addEventListener()` with an event handler registered on an object property like `onclick`.

15.2.4 Event Propagation

When the target of an event is the Window object or some other standalone object, the browser responds to an event simply by invoking the appropriate handlers on that one object. When the event target is a Document or document Element, however, the situation is more complicated.

After the event handlers registered on the target element are invoked, most events "bubble" up the DOM tree. The event handlers of the target's parent are invoked. Then the handlers registered on the target's grandparent are invoked. This continues up to the Document object, and then beyond to the Window object. Event bubbling provides an alternative to registering handlers on lots of individual document elements: instead, you can register a single handler on a common ancestor element and handle events there. You might register a "change" handler on a `<form>` element, for example, instead of registering a "change" handler for every element in the form.

Most events that occur on document elements bubble. Notable exceptions are the "focus," "blur," and "scroll" events. The "load" event on document elements bubbles, but it stops bubbling at the Document object and does not propagate on to the Window object. (The "load" event handlers of the Window object are triggered only when the entire document has loaded.)

Event bubbling is the third "phase" of event propagation. The invocation of the event handlers of the target object itself is the second phase. The first phase, which occurs even before the target handlers are invoked, is called the "capturing" phase. Recall that `addEventListener()` takes an optional third argument. If that argument is `true`,

or `{capture:true}`, then the event handler is registered as a capturing event handler for invocation during this first phase of event propagation. The capturing phase of event propagation is like the bubbling phase in reverse. The capturing handlers of the Window object are invoked first, then the capturing handlers of the Document object, then of the body object, and so on down the DOM tree until the capturing event handlers of the parent of the event target are invoked. Capturing event handlers registered on the event target itself are not invoked.

Event capturing provides an opportunity to peek at events before they are delivered to their target. A capturing event handler can be used for debugging, or it can be used along with the event cancellation technique described in the next section to filter events so that the target event handlers are never actually invoked. One common use for event capturing is handling mouse drags, where mouse motion events need to be handled by the object being dragged, not the document elements over which it is dragged.

15.2.5 Event Cancellation

Browsers respond to many user events, even if your code does not: when the user clicks the mouse on a hyperlink, the browser follows the link. If an HTML text input element has the keyboard focus and the user types a key, the browser will enter the user's input. If the user moves their finger across a touch-screen device, the browser scrolls. If you register an event handler for events like these, you can prevent the browser from performing its default action by invoking the `preventDefault()` method of the event object. (Unless you registered the handler with the `passive` option, which makes `preventDefault()` ineffective.)

Canceling the default action associated with an event is only one kind of event cancellation. We can also cancel the propagation of events by calling the `stopPropagation()` method of the event object. If there are other handlers defined on the same object, the rest of those handlers will still be invoked, but no event handlers on any other object will be invoked after `stopPropagation()` is called. `stopPropagation()` works during the capturing phase, at the event target itself, and during the bubbling phase. `stopImmediatePropagation()` works like `stopPropagation()`, but it also prevents the invocation of any subsequent event handlers registered on the same object.

15.2.6 Dispatching Custom Events

Client-side JavaScript's event API is a relatively powerful one, and you can use it to define and dispatch your own events. Suppose, for example, that your program periodically needs to perform a long calculation or make a network request and that, while this operation is pending, other operations are not possible. You want to let the user know about this by displaying "spinners" to indicate that the application is busy. But the module that is busy should not need to know where the spinners should be

displayed. Instead, that module might just dispatch an event to announce that it is busy and then dispatch another event when it is no longer busy. Then, the UI module can register event handlers for those events and take whatever UI actions are appropriate to notify the user.

If a JavaScript object has an `addEventListener()` method, then it is an "event target," and this means it also has a `dispatchEvent()` method. You can create your own event object with the `CustomEvent()` constructor and pass it to `dispatchEvent()`. The first argument to `CustomEvent()` is a string that specifies the type of your event, and the second argument is an object that specifies the properties of the event object. Set the `detail` property of this object to a string, object, or other value that represents the content of your event. If you plan to dispatch your event on a document element and want it to bubble up the document tree, add `bubbles:true` to the second argument:

```
// Dispatch a custom event so the UI knows we are busy
document.dispatchEvent(new CustomEvent("busy", { detail: true }));

// Perform a network operation
fetch(url)
  .then(handleNetworkResponse)
  .catch(handleNetworkError)
  .finally(() => {
      // After the network request has succeeded or failed, dispatch
      // another event to let the UI know that we are no longer busy.
      document.dispatchEvent(new CustomEvent("busy", { detail: false }));
  });

// Elsewhere, in your program you can register a handler for "busy" events
// and use it to show or hide the spinner to let the user know.
document.addEventListener("busy", (e) => {
    if (e.detail) {
        showSpinner();
    } else {
        hideSpinner();
    }
});
```

15.3 Scripting Documents

Client-side JavaScript exists to turn static HTML documents into interactive web applications. So scripting the content of web pages is really the central purpose of JavaScript.

Every Window object has a `document` property that refers to a Document object. The Document object represents the content of the window, and it is the subject of this section. The Document object does not stand alone, however. It is the central object in the DOM for representing and manipulating document content.

The DOM was introduced in §15.1.2. This section explains the API in detail. It covers:

- How to query or *select* individual elements from a document.
- How to *traverse* a document, and how to find the ancestors, siblings, and descendants of any document element.
- How to query and set the attributes of document elements.
- How to query, set, and modify the content of a document.
- How to modify the structure of a document by creating, inserting, and deleting nodes.

15.3.1 Selecting Document Elements

Client-side JavaScript programs often need to manipulate one or more elements within the document. The global `document` property refers to the Document object, and the Document object has `head` and `body` properties that refer to the Element objects for the `<head>` and `<body>` tags, respectively. But a program that wants to manipulate an element embedded more deeply in the document must somehow obtain or *select* the Element objects that refer to those document elements.

Selecting elements with CSS selectors

CSS stylesheets have a very powerful syntax, known as *selectors*, for describing elements or sets of elements within a document. The DOM methods `querySelector()` and `querySelectorAll()` allow us to find the element or elements within a document that match a specified CSS selector. Before we cover the methods, we'll start with a quick tutorial on CSS selector syntax.

CSS selectors can describe elements by tag name, the value of their `id` attribute, or the words in their `class` attribute:

```
div                     // Any <div> element
#nav                    // The element with id="nav"
.warning                // Any element with "warning" in its class attribute
```

The `#` character is used to match based on the `id` attribute, and the `.` character is used to match based on the `class` attribute. Elements can also be selected based on more general attribute values:

```
p[lang="fr"]            // A paragraph written in French: <p lang="fr">
*[name="x"]             // Any element with a name="x" attribute
```

Note that these examples combine a tag name selector (or the `*` tag name wildcard) with an attribute selector. More complex combinations are also possible:

```
span.fatal.error        // Any <span> with "fatal" and "error" in its class
span[lang="fr"].warning // Any <span> in French with class "warning"
```

Selectors can also specify document structure:

```
#log span               // Any <span> descendant of the element with id="log"
#log>span               // Any <span> child of the element with id="log"
body>h1:first-child     // The first <h1> child of the <body>
img + p.caption         // A <p> with class "caption" immediately after an <img>
h2 ~ p                  // Any <p> that follows an <h2> and is a sibling of it
```

If two selectors are separated by a comma, it means that we've selected elements that match either one of the selectors:

```
button, input[type="button"] // All <button> and <input type="button"> elements
```

As you can see, CSS selectors allow us to refer to elements within a document by type, ID, class, attributes, and position within the document. The `querySelector()` method takes a CSS selector string as its argument and returns the first matching element in the document that it finds, or returns `null` if none match:

```
// Find the document element for the HTML tag with attribute id="spinner"
let spinner = document.querySelector("#spinner");
```

`querySelectorAll()` is similar, but it returns all matching elements in the document rather than just returning the first:

```
// Find all Element objects for <h1>, <h2>, and <h3> tags
let titles = document.querySelectorAll("h1, h2, h3");
```

The return value of `querySelectorAll()` is not an array of Element objects. Instead, it is an array-like object known as a NodeList. NodeList objects have a `length` property and can be indexed like arrays, so you can loop over them with a traditional `for` loop. NodeLists are also iterable, so you can use them with `for/of` loops as well. If you want to convert a NodeList into a true array, simply pass it to `Array.from()`.

The NodeList returned by `querySelectorAll()` will have a `length` property set to 0 if there are not any elements in the document that match the specified selector.

`querySelector()` and `querySelectorAll()` are implemented by the Element class as well as by the Document class. When invoked on an element, these methods will only return elements that are descendants of that element.

Note that CSS defines `::first-line` and `::first-letter` pseudoelements. In CSS, these match portions of text nodes rather than actual elements. They will not match if used with `querySelectorAll()` or `querySelector()`. Also, many browsers will refuse to return matches for the `:link` and `:visited` pseudoclasses, as this could expose information about the user's browsing history.

Another CSS-based element selection method is `closest()`. This method is defined by the Element class and takes a selector as its only argument. If the selector matches

the element it is invoked on, it returns that element. Otherwise, it returns the closest ancestor element that the selector matches, or returns `null` if none matched. In a sense, `closest()` is the opposite of `querySelector()`: `closest()` starts at an element and looks for a match above it in the tree, while `querySelector()` starts with an element and looks for a match below it in the tree. `closest()` can be useful when you have registered an event handler at a high level in the document tree. If you are handling a "click" event, for example, you might want to know whether it is a click a hyperlink. The event object will tell you what the target was, but that target might be the text inside a link rather than the hyperlink's `<a>` tag itself. Your event handler could look for the nearest containing hyperlink like this:

```
// Find the closest enclosing <a> tag that has an href attribute.
let hyperlink = event.target.closest("a[href]");
```

Here is another way you might use `closest()`:

```
// Return true if the element e is inside of an HTML list element
function insideList(e) {
    return e.closest("ul,ol,dl") !== null;
}
```

The related method `matches()` does not return ancestors or descendants: it simply tests whether an element is matched by a CSS selector and returns `true` if so and `false` otherwise:

```
// Return true if e is an HTML heading element
function isHeading(e) {
    return e.matches("h1,h2,h3,h4,h5,h6");
}
```

Other element selection methods

In addition to `querySelector()` and `querySelectorAll()`, the DOM also defines a number of older element selection methods that are more or less obsolete now. You may still see some of these methods (especially `getElementById()`) in use, however:

```
// Look up an element by id. The argument is just the id, without
// the CSS selector prefix #. Similar to document.querySelector("#sect1")
let sect1 = document.getElementById("sect1");

// Look up all elements (such as form checkboxes) that have a name="color"
// attribute. Similar to document.querySelectorAll('*[name="color"]');
let colors = document.getElementsByName("color");

// Look up all <h1> elements in the document.
// Similar to document.querySelectorAll("h1")
let headings = document.getElementsByTagName("h1");

// getElementsByTagName() is also defined on elements.
// Get all <h2> elements within the sect1 element.
```

```
let subheads = sect1.getElementsByTagName("h2");

// Look up all elements that have class "tooltip."
// Similar to document.querySelectorAll(".tooltip")
let tooltips = document.getElementsByClassName("tooltip");

// Look up all descendants of sect1 that have class "sidebar"
// Similar to sect1.querySelectorAll(".sidebar")
let sidebars = sect1.getElementsByClassName("sidebar");
```

Like `querySelectorAll()`, the methods in this code return a NodeList (except for `getElementById()`, which returns a single Element object). Unlike `querySelectorAll()`, however, the NodeLists returned by these older selection methods are "live," which means that the length and content of the list can change if the document content or structure changes.

Preselected elements

For historical reasons, the Document class defines shortcut properties to access certain kinds of nodes. The `images`, `forms`, and `links` properties, for example, provide easy access to the `<img>`, `<form>`, and `<a>` elements (but only `<a>` tags that have an `href` attribute) of a document. These properties refer to HTMLCollection objects, which are much like NodeList objects, but they can additionally be indexed by element ID or name. With the `document.forms` property, for example, you can access the `<form id="address">` tag as:

```
document.forms.address;
```

An even more outdated API for selecting elements is the `document.all` property, which is like an HTMLCollection for all elements in the document. `document.all` is deprecated, and you should no longer use it.

15.3.2 Document Structure and Traversal

Once you have selected an Element from a Document, you sometimes need to find structurally related portions (parent, siblings, children) of the document. When we are primarily interested in the Elements of a document instead of the text within them (and the whitespace between them, which is also text), there is a traversal API that allows us to treat a document as a tree of Element objects, ignoring Text nodes that are also part of the document. This traversal API does not involve any methods; it is simply a set of properties on Element objects that allow us to refer to the parent, children, and siblings of a given element:

`parentNode`
: This property of an element refers to the parent of the element, which will be another Element or a Document object.

`children`
: This NodeList contains the Element children of an element, but excludes non-Element children like Text nodes (and Comment nodes).

`childElementCount`
: The number of Element children. Returns the same value as `children.length`.

`firstElementChild`, `lastElementChild`
: These properties refer to the first and last Element children of an Element. They are `null` if the Element has no Element children.

`nextElementSibling`, `previousElementSibling`
: These properties refer to the sibling Elements immediately before or immediately after an Element, or `null` if there is no such sibling.

Using these Element properties, the second child Element of the first child Element of the Document can be referred to with either of these expressions:

```
document.children[0].children[1]
document.firstElementChild.firstElementChild.nextElementSibling
```

(In a standard HTML document, both of those expressions refer to the `<body>` tag of the document.)

Here are two functions that demonstrate how you can use these properties to recursively do a depth-first traversal of a document invoking a specified function for every element in the document:

```
// Recursively traverse the Document or Element e, invoking the function
// f on e and on each of its descendants
function traverse(e, f) {
    f(e);                           // Invoke f() on e
    for(let child of e.children) {  // Iterate over the children
        traverse(child, f);         // And recurse on each one
    }
}

function traverse2(e, f) {
    f(e);                           // Invoke f() on e
    let child = e.firstElementChild; // Iterate the children linked-list style
    while(child !== null) {
        traverse2(child, f);        // And recurse
        child = child.nextElementSibling;
    }
}
```

Documents as trees of nodes

If you want to traverse a document or some portion of a document and do not want to ignore the Text nodes, you can use a different set of properties defined on all Node objects. This will allow you to see Elements, Text nodes, and even Comment nodes (which represent HTML comments in the document).

All Node objects define the following properties:

`parentNode`
: The node that is the parent of this one, or `null` for nodes like the Document object that have no parent.

`childNodes`
: A read-only NodeList that that contains all children (not just Element children) of the node.

`firstChild`, `lastChild`
: The first and last child nodes of a node, or `null` if the node has no children.

`nextSibling`, `previousSibling`
: The next and previous sibling nodes of a node. These properties connect nodes in a doubly linked list.

`nodeType`
: A number that specifies what kind of node this is. Document nodes have value 9. Element nodes have value 1. Text nodes have value 3. Comment nodes have value 8.

`nodeValue`
: The textual content of a Text or Comment node.

`nodeName`
: The HTML tag name of an Element, converted to uppercase.

Using these Node properties, the second child node of the first child of the Document can be referred to with expressions like these:

```
document.childNodes[0].childNodes[1]
document.firstChild.firstChild.nextSibling
```

Suppose the document in question is the following:

```
<html><head><title>Test</title></head><body>Hello World!</body></html>
```

Then the second child of the first child is the `<body>` element. It has a `nodeType` of 1 and a `nodeName` of "BODY".

Note, however, that this API is extremely sensitive to variations in the document text. If the document is modified by inserting a single newline between the `<html>` and the

<head> tag, for example, the Text node that represents that newline becomes the first child of the first child, and the second child is the <head> element instead of the <body> element.

To demonstrate this Node-based traversal API, here is a function that returns all of the text within an element or document:

```
// Return the plain-text content of element e, recursing into child elements.
// This method works like the textContent property
function textContent(e) {
    let s = "";                         // Accumulate the text here
    for(let child = e.firstChild; child !== null; child = child.nextSibling) {
        let type = child.nodeType;
        if (type === 3) {               // If it is a Text node
            s += child.nodeValue;       // add the text content to our string.
        } else if (type === 1) {        // And if it is an Element node
            s += textContent(child);    // then recurse.
        }
    }
    return s;
}
```

This function is a demonstration only—in practice, you would simply write `e.textContent` to obtain the textual content of the element `e`.

15.3.3 Attributes

HTML elements consist of a tag name and a set of name/value pairs known as *attributes*. The <a> element that defines a hyperlink, for example, uses the value of its `href` attribute as the destination of the link.

The Element class defines general `getAttribute()`, `setAttribute()`, `hasAttribute()`, and `removeAttribute()` methods for querying, setting, testing, and removing the attributes of an element. But the attribute values of HTML elements (for all standard attributes of standard HTML elements) are available as properties of the HTMLElement objects that represent those elements, and it is usually much easier to work with them as JavaScript properties than it is to call `getAttribute()` and related methods.

HTML attributes as element properties

The Element objects that represent the elements of an HTML document usually define read/write properties that mirror the HTML attributes of the elements. Element defines properties for the universal HTML attributes such as `id`, `title`, `lang`, and `dir` and event handler properties like `onclick`. Element-specific subtypes define attributes specific to those elements. To query the URL of an image, for example, you can use the `src` property of the HTMLElement that represents the <img> element:

```
let image = document.querySelector("#main_image");
let url = image.src;       // The src attribute is the URL of the image
image.id === "main_image"  // => true; we looked up the image by id
```

Similarly, you might set the form-submission attributes of a `<form>` element with code like this:

```
let f = document.querySelector("form");      // First <form> in the document
f.action = "https://www.example.com/submit"; // Set the URL to submit it to.
f.method = "POST";                           // Set the HTTP request type.
```

For some elements, such as the `<input>` element, some HTML attribute names map to differently named properties. The HTML `value` attribute of an `<input>`, for example, is mirrored by the JavaScript `defaultValue` property. The JavaScript `value` property of the `<input>` element contains the user's current input, but changes to the `value` property do not affect the `defaultValue` property nor the `value` attribute.

HTML attributes are not case sensitive, but JavaScript property names are. To convert an attribute name to the JavaScript property, write it in lowercase. If the attribute is more than one word long, however, put the first letter of each word after the first in uppercase: `defaultChecked` and `tabIndex`, for example. Event handler properties like `onclick` are an exception, however, and are written in lowercase.

Some HTML attribute names are reserved words in JavaScript. For these, the general rule is to prefix the property name with "html". The HTML `for` attribute (of the `<label>` element), for example, becomes the JavaScript `htmlFor` property. "class" is a reserved word in JavaScript, and the very important HTML `class` attribute is an exception to the rule: it becomes `className` in JavaScript code.

The properties that represent HTML attributes usually have string values. But when the attribute is a boolean or numeric value (the `defaultChecked` and `maxLength` attributes of an `<input>` element, for example), the properties are booleans or numbers instead of strings. Event handler attributes always have functions (or `null`) as their values.

Note that this property-based API for getting and setting attribute values does not define any way to remove an attribute from an element. In particular, the `delete` operator cannot be used for this purpose. If you need to delete an attribute, use the `removeAttribute()` method.

The class attribute

The `class` attribute of an HTML element is a particularly important one. Its value is a space-separated list of CSS classes that apply to the element and affect how it is styled with CSS. Because `class` is a reserved word in JavaScript, the value of this attribute is available through the `className` property on Element objects. The `className` property can set and return the value of the `class` attribute as a string. But the `class`

attribute is poorly named: its value is a list of CSS classes, not a single class, and it is common in client-side JavaScript programming to want to add and remove individual class names from this list rather than work with the list as a single string.

For this reason, Element objects define a `classList` property that allows you to treat the `class` attribute as a list. The value of the `classList` property is an iterable Array-like object. Although the name of the property is `classList`, it behaves more like a set of classes, and defines `add()`, `remove()`, `contains()`, and `toggle()` methods:

```
// When we want to let the user know that we are busy, we display
// a spinner. To do this we have to remove the "hidden" class and add the
// "animated" class (assuming the stylesheets are configured correctly).
let spinner = document.querySelector("#spinner");
spinner.classList.remove("hidden");
spinner.classList.add("animated");
```

Dataset attributes

It is sometimes useful to attach additional information to HTML elements, typically when JavaScript code will be selecting those elements and manipulating them in some way. In HTML, any attribute whose name is lowercase and begins with the prefix "data-" is considered valid, and you can use them for any purpose. These "dataset attributes" will not affect the presentation of the elements on which they appear, and they define a standard way to attach additional data without compromising document validity.

In the DOM, Element objects have a `dataset` property that refers to an object that has properties that correspond to the `data-` attributes with their prefix removed. Thus, `dataset.x` would hold the value of the `data-x` attribute. Hyphenated attributes map to camelCase property names: the attribute `data-section-number` becomes the property `dataset.sectionNumber`.

Suppose an HTML document contains this text:

```
<h2 id="title" data-section-number="16.1">Attributes</h2>
```

Then you could write JavaScript like this to access that section number:

```
let number = document.querySelector("#title").dataset.sectionNumber;
```

15.3.4 Element Content

Look again at the document tree pictured in Figure 15-1, and ask yourself what the "content" of the <p> element is. There are two ways we might answer this question:

- The content is the HTML string "This is a simple document".
- The content is the plain-text string "This is a simple document".

Both of these are valid answers, and each answer is useful in its own way. The sections that follow explain how to work with the HTML representation and the plain-text representation of an element's content.

Element content as HTML

Reading the `innerHTML` property of an Element returns the content of that element as a string of markup. Setting this property on an element invokes the web browser's parser and replaces the element's current content with a parsed representation of the new string. You can test this out by opening the developer console and typing:

```
document.body.innerHTML = "<h1>Oops</h1>";
```

You will see that the entire web page disappears and is replaced with the single heading, "Oops". Web browsers are very good at parsing HTML, and setting `innerHTML` is usually fairly efficient. Note, however, that appending text to the `innerHTML` property with the `+=` operator is not efficient because it requires both a serialization step to convert element content to a string and then a parsing step to convert the new string back into element content.

When using these HTML APIs, it is very important that you never insert user input into the document. If you do this, you allow malicious users to inject their own scripts into your application. See "Cross-site scripting" on page 425 for details.

The `outerHTML` property of an Element is like `innerHTML` except that its value includes the element itself. When you query `outerHTML`, the value includes the opening and closing tags of the element. And when you set `outerHTML` on an element, the new content replaces the element itself.

A related Element method is `insertAdjacentHTML()`, which allows you to insert a string of arbitrary HTML markup "adjacent" to the specified element. The markup is passed as the second argument to this method, and the precise meaning of "adjacent" depends on the value of the first argument. This first argument should be a string with one of the values "beforebegin," "afterbegin," "beforeend," or "afterend." These values correspond to insertion points that are illustrated in Figure 15-2.

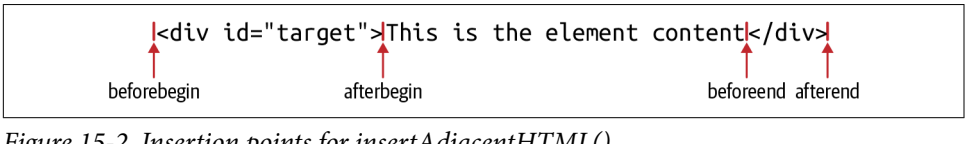

Figure 15-2. Insertion points for insertAdjacentHTML()

Element content as plain text

Sometimes you want to query the content of an element as plain text or to insert plain text into a document (without having to escape the angle brackets and ampersands used in HTML markup). The standard way to do this is with the `textContent` property:

```
let para = document.querySelector("p"); // First <p> in the document
let text = para.textContent;            // Get the text of the paragraph
para.textContent = "Hello World!";      // Alter the text of the paragraph
```

The `textContent` property is defined by the Node class, so it works for Text nodes as well as Element nodes. For Element nodes, it finds and returns all text in all descendants of the element.

The Element class defines an `innerText` property that is similar to `textContent`. `innerText` has some unusual and complex behaviors, such as attempting to preserve table formatting. It is not well specified nor implemented compatibly between browsers, however, and should no longer be used.

Text in <script> Elements

Inline `<script>` elements (i.e., those that do not have a `src` attribute) have a `text` property that you can use to retrieve their text. The content of a `<script>` element is never displayed by the browser, and the HTML parser ignores angle brackets and ampersands within a script. This makes a `<script>` element an ideal place to embed arbitrary textual data for use by your application. Simply set the `type` attribute of the element to some value (such as "text/x-custom-data") that makes it clear that the script is not executable JavaScript code. If you do this, the JavaScript interpreter will ignore the script, but the element will exist in the document tree, and its `text` property will return the data to you.

15.3.5 Creating, Inserting, and Deleting Nodes

We've seen how to query and alter document content using strings of HTML and of plain text. And we've also seen that we can traverse a Document to examine the individual Element and Text nodes that it is made of. It is also possible to alter a document at the level of individual nodes. The Document class defines methods for creating Element objects, and Element and Text objects have methods for inserting, deleting, and replacing nodes in the tree.

Create a new element with the `createElement()` method of the Document class and append strings of text or other elements to it with its `append()` and `prepend()` methods:

```
let paragraph = document.createElement("p"); // Create an empty <p> element
let emphasis = document.createElement("em"); // Create an empty <em> element
emphasis.append("World");                    // Add text to the <em> element
paragraph.append("Hello ", emphasis, "!");   // Add text and <em> to <p>
paragraph.prepend("¡");                      // Add more text at start of <p>
paragraph.innerHTML                          // => "¡Hello <em>World</em>!"
```

`append()` and `prepend()` take any number of arguments, which can be Node objects or strings. String arguments are automatically converted to Text nodes. (You can create Text nodes explicitly with `document.createTextNode()`, but there is rarely any reason to do so.) `append()` adds the arguments to the element at the end of the child list. `prepend()` adds the arguments at the start of the child list.

If you want to insert an Element or Text node into the middle of the containing element's child list, then neither `append()` or `prepend()` will work for you. In this case, you should obtain a reference to a sibling node and call `before()` to insert the new content before that sibling or `after()` to insert it after that sibling. For example:

```
// Find the heading element with class="greetings"
let greetings = document.querySelector("h2.greetings");

// Now insert the new paragraph and a horizontal rule after that heading
greetings.after(paragraph, document.createElement("hr"));
```

Like `append()` and `prepend()`, `after()` and `before()` take any number of string and element arguments and insert them all into the document after converting strings to Text nodes. `append()` and `prepend()` are only defined on Element objects, but `after()` and `before()` work on both Element and Text nodes: you can use them to insert content relative to a Text node.

Note that elements can only be inserted at one spot in the document. If an element is already in the document and you insert it somewhere else, it will be moved to the new location, not copied:

```
// We inserted the paragraph after this element, but now we
// move it so it appears before the element instead
greetings.before(paragraph);
```

If you do want to make a copy of an element, use the `cloneNode()` method, passing `true` to copy all of its content:

```
// Make a copy of the paragraph and insert it after the greetings element
greetings.after(paragraph.cloneNode(true));
```

You can remove an Element or Text node from the document by calling its `remove()` method, or you can replace it by calling `replaceWith()` instead. `remove()` takes no arguments, and `replaceWith()` takes any number of strings and elements just like `before()` and `after()` do:

```
// Remove the greetings element from the document and replace it with
// the paragraph element (moving the paragraph from its current location
// if it is already inserted into the document).
greetings.replaceWith(paragraph);

// And now remove the paragraph.
paragraph.remove();
```

The DOM API also defines an older generation of methods for inserting and removing content. `appendChild()`, `insertBefore()`, `replaceChild()`, and `removeChild()` are harder to use than the methods shown here and should never be needed.

15.3.6 Example: Generating a Table of Contents

Example 15-1 shows how to dynamically create a table of contents for a document. It demonstrates many of the document scripting techniques described in the previous sections. The example is well commented, and you should have no trouble following the code.

Example 15-1. Generating a table of contents with the DOM API

```
/**
 * TOC.js: create a table of contents for a document.
 *
 * This script runs when the DOMContentLoaded event is fired and
 * automatically generates a table of contents for the document.
 * It does not define any global symbols so it should not conflict
 * with other scripts.
 *
 * When this script runs, it first looks for a document element with
 * an id of "TOC". If there is no such element it creates one at the
 * start of the document. Next, the function finds all <h2> through
 * <h6> tags, treats them as section titles, and creates a table of
 * contents within the TOC element. The function adds section numbers
 * to each section heading and wraps the headings in named anchors so
 * that the TOC can link to them. The generated anchors have names
 * that begin with "TOC", so you should avoid this prefix in your own
 * HTML.
 *
 * The entries in the generated TOC can be styled with CSS. All
 * entries have a class "TOCEntry". Entries also have a class that
 * corresponds to the level of the section heading. <h1> tags generate
 * entries of class "TOCLevel1", <h2> tags generate entries of class
 * "TOCLevel2", and so on. Section numbers inserted into headings have
 * class "TOCSectNum".
 *
 * You might use this script with a stylesheet like this:
 *
 *   #TOC { border: solid black 1px; margin: 10px; padding: 10px; }
 *   .TOCEntry { margin: 5px 0px; }
```

```
 *   .TOCEntry a { text-decoration: none; }
 *   .TOCLevel1 { font-size: 16pt; font-weight: bold; }
 *   .TOCLevel2 { font-size: 14pt; margin-left: .25in; }
 *   .TOCLevel3 { font-size: 12pt; margin-left: .5in; }
 *   .TOCSectNum:after { content: ": "; }
 *
 * To hide the section numbers, use this:
 *
 *   .TOCSectNum { display: none }
 **/
document.addEventListener("DOMContentLoaded", () => {
    // Find the TOC container element.
    // If there isn't one, create one at the start of the document.
    let toc = document.querySelector("#TOC");
    if (!toc) {
        toc = document.createElement("div");
        toc.id = "TOC";
        document.body.prepend(toc);
    }

    // Find all section heading elements. We're assuming here that the
    // document title uses <h1> and that sections within the document are
    // marked with <h2> through <h6>.
    let headings = document.querySelectorAll("h2,h3,h4,h5,h6");

    // Initialize an array that keeps track of section numbers.
    let sectionNumbers = [0,0,0,0,0];

    // Now loop through the section header elements we found.
    for(let heading of headings) {
        // Skip the heading if it is inside the TOC container.
        if (heading.parentNode === toc) {
            continue;
        }

        // Figure out what level heading it is.
        // Subtract 1 because <h2> is a level-1 heading.
        let level = parseInt(heading.tagName.charAt(1)) - 1;

        // Increment the section number for this heading level
        // and reset all lower heading level numbers to zero.
        sectionNumbers[level-1]++;
        for(let i = level; i < sectionNumbers.length; i++) {
            sectionNumbers[i] = 0;
        }

        // Now combine section numbers for all heading levels
        // to produce a section number like 2.3.1.
        let sectionNumber = sectionNumbers.slice(0, level).join(".");

        // Add the section number to the section header title.
        // We place the number in a <span> to make it styleable.
```

```
            let span = document.createElement("span");
            span.className = "TOCSectNum";
            span.textContent = sectionNumber;
            heading.prepend(span);

            // Wrap the heading in a named anchor so we can link to it.
            let anchor = document.createElement("a");
            let fragmentName = `TOC${sectionNumber}`;
            anchor.name = fragmentName;
            heading.before(anchor);    // Insert anchor before heading
            anchor.append(heading);    // and move heading inside anchor

            // Now create a link to this section.
            let link = document.createElement("a");
            link.href = `#${fragmentName}`;     // Link destination

            // Copy the heading text into the link. This is a safe use of
            // innerHTML because we are not inserting any untrusted strings.
            link.innerHTML = heading.innerHTML;

            // Place the link in a div that is styleable based on the level.
            let entry = document.createElement("div");
            entry.classList.add("TOCEntry", `TOCLevel${level}`);
            entry.append(link);

            // And add the div to the TOC container.
            toc.append(entry);
        }
    });
```

15.4 Scripting CSS

We've seen that JavaScript can control the logical structure and content of HTML documents. It can also control the visual appearance and layout of those documents by scripting CSS. The following subsections explain a few different techniques that JavaScript code can use to work with CSS.

This is a book about JavaScript, not about CSS, and this section assumes that you already have a working knowledge of how CSS is used to style HTML content. But it's worth mentioning some of the CSS styles that are commonly scripted from JavaScript:

- Setting the `display` style to "none" hides an element. You can later show the element by setting `display` to some other value.
- You can dynamically position elements by setting the `position` style to "absolute," "relative," or "fixed" and then setting the `top` and `left` styles to the desired coordinates. This is important when using JavaScript to display dynamic content like modal dialogues and tooltips.

- You can shift, scale, and rotate elements with the `transform` style.
- You can animate changes to other CSS styles with the `transition` style. These animations are handled automatically by the web browser and do not require JavaScript, but you can use JavaScript to initiate the animations.

15.4.1 CSS Classes

The simplest way to use JavaScript to affect the styling of document content is to add and remove CSS class names from the `class` attribute of HTML tags. This is easy to do with the `classList` property of Element objects, as explained in "The class attribute" on page 445.

Suppose, for example, that your document's stylesheet includes a definition for a "hidden" class:

```
.hidden {
  display:none;
}
```

With this style defined, you can hide (and then show) an element with code like this:

```
// Assume that this "tooltip" element has class="hidden" in the HTML file.
// We can make it visible like this:
document.querySelector("#tooltip").classList.remove("hidden");

// And we can hide it again like this:
document.querySelector("#tooltip").classList.add("hidden");
```

15.4.2 Inline Styles

To continue with the preceding tooltip example, suppose that the document is structured with only a single tooltip element, and we want to dynamically position it before displaying it. In general, we can't create a different stylesheet class for each possible position of the tooltip, so the `classList` property won't help us with positioning.

In this case, we need to script the `style` attribute of the tooltip element to set inline styles that are specific to that one element. The DOM defines a `style` property on all Element objects that correspond to the `style` attribute. Unlike most such properties, however, the `style` property is not a string. Instead, it is a CSSStyleDeclaration object: a parsed representation of the CSS styles that appear in textual form in the `style` attribute. To display and set the position of our hypothetical tooltip with JavaScript, we might use code like this:

```
function displayAt(tooltip, x, y) {
    tooltip.style.display = "block";
    tooltip.style.position = "absolute";
```

```
    tooltip.style.left = `${x}px`;
    tooltip.style.top = `${y}px`;
}
```

Naming Conventions: CSS Properties in JavaScript

Many CSS style properties, such as `font-size`, contain hyphens in their names. In JavaScript, a hyphen is interpreted as a minus sign and is not allowed in property names or other identifiers. Therefore, the names of the properties of the CSSStyleDeclaration object are slightly different from the names of actual CSS properties. If a CSS property name contains one or more hyphens, the CSSStyleDeclaration property name is formed by removing the hyphens and capitalizing the letter immediately following each hyphen. The CSS property `border-left-width` is accessed through the JavaScript `borderLeftWidth` property, for example, and the CSS `font-family` property is written as `fontFamily` in JavaScript.

When working with the style properties of the CSSStyleDeclaration object, remember that all values must be specified as strings. In a stylesheet or `style` attribute, you can write:

```
display: block; font-family: sans-serif; background-color: #ffffff;
```

To accomplish the same thing for an element `e` with JavaScript, you have to quote all of the values:

```
e.style.display = "block";
e.style.fontFamily = "sans-serif";
e.style.backgroundColor = "#ffffff";
```

Note that the semicolons go outside the strings. These are just normal JavaScript semicolons; the semicolons you use in CSS stylesheets are not required as part of the string values you set with JavaScript.

Furthermore, remember that many CSS properties require units such as "px" for pixels or "pt" for points. Thus, it is not correct to set the `marginLeft` property like this:

```
e.style.marginLeft = 300;    // Incorrect: this is a number, not a string
e.style.marginLeft = "300";  // Incorrect: the units are missing
```

Units are required when setting style properties in JavaScript, just as they are when setting style properties in stylesheets. The correct way to set the value of the `marginLeft` property of an element `e` to 300 pixels is:

```
e.style.marginLeft = "300px";
```

If you want to set a CSS property to a computed value, be sure to append the units at the end of the computation:

```
e.style.left = `${x0 + left_border + left_padding}px`;
```

Recall that some CSS properties, such as `margin`, are shortcuts for other properties, such as `margin-top`, `margin-right`, `margin-bottom`, and `margin-left`. The CSSStyleDeclaration object has properties that correspond to these shortcut properties. For example, you might set the `margin` property like this:

```
e.style.margin = `${top}px ${right}px ${bottom}px ${left}px`;
```

Sometimes, you may find it easier to set or query the inline style of an element as a single string value rather than as a CSSStyleDeclaration object. To do that, you can use the Element `getAttribute()` and `setAttribute()` methods, or you can use the `cssText` property of the CSSStyleDeclaration object:

```
// Copy the inline styles of element e to element f:
f.setAttribute("style", e.getAttribute("style"));

// Or do it like this:
f.style.cssText = e.style.cssText;
```

When querying the `style` property of an element, keep in mind that it represents only the inline styles of an element and that most styles for most elements are specified in stylesheets rather than inline. Furthermore, the values you obtain when querying the `style` property will use whatever units and whatever shortcut property format is actually used on the HTML attribute, and your code may have to do some sophisticated parsing to interpret them. In general, if you want to query the styles of an element, you probably want the *computed style*, which is discussed next.

15.4.3 Computed Styles

The computed style for an element is the set of property values that the browser derives (or computes) from the element's inline style plus all applicable style rules in all stylesheets: it is the set of properties actually used to display the element. Like inline styles, computed styles are represented with a CSSStyleDeclaration object. Unlike inline styles, however, computed styles are read-only. You can't set these styles, but the computed CSSStyleDeclaration object for an element lets you determine what style property values the browser used when rendering that element.

Obtain the computed style for an element with the `getComputedStyle()` method of the Window object. The first argument to this method is the element whose computed style is desired. The optional second argument is used to specify a CSS pseudoelement, such as "::before" or "::after":

```
let title = document.querySelector("#section1title");
let styles = window.getComputedStyle(title);
let beforeStyles = window.getComputedStyle(title, "::before");
```

The return value of `getComputedStyle()` is a CSSStyleDeclaration object that represents all the styles that apply to the specified element (or pseudoelement). There are a

number of important differences between a CSSStyleDeclaration object that represents inline styles and one that represents computed styles:

- Computed style properties are read-only.
- Computed style properties are *absolute*: relative units like percentages and points are converted to absolute values. Any property that specifies a size (such as a margin size or a font size) will have a value measured in pixels. This value will be a string with a "px" suffix, so you'll still need to parse it, but you won't have to worry about parsing or converting other units. Properties whose values are colors will be returned in "rgb()" or "rgba()" format.
- Shortcut properties are not computed—only the fundamental properties that they are based on are. Don't query the `margin` property, for example, but use `marginLeft`, `marginTop`, and so on. Similarly, don't query `border` or even `borderWidth`. Instead, use `borderLeftWidth`, `borderTopWidth`, and so on.
- The `cssText` property of the computed style is undefined.

A CSSStyleDeclaration object returned by `getComputedStyle()` generally contains much more information about an element than the CSSStyleDeclaration obtained from the inline `style` property of that element. But computed styles can be tricky, and querying them does not always provide the information you might expect. Consider the `font-family` attribute: it accepts a comma-separated list of desired font families for cross-platform portability. When you query the `fontFamily` property of a computed style, you're simply getting the value of the most specific `font-family` style that applies to the element. This may return a value such as "arial,helvetica,sans-serif," which does not tell you which typeface is actually in use. Similarly, if an element is not absolutely positioned, attempting to query its position and size through the `top` and `left` properties of its computed style often returns the value `auto`. This is a perfectly legal CSS value, but it is probably not what you were looking for.

Although CSS can be used to precisely specify the position and size of document elements, querying the computed style of an element is not the preferred way to determine the element's size and position. See §15.5.2 for a simpler, portable alternative.

15.4.4 Scripting Stylesheets

In addition to scripting class attributes and inline styles, JavaScript can also manipulate stylesheets themselves. Stylesheets are associated with an HTML document with a `<style>` tag or with a `<link rel="stylesheet">` tag. Both of these are regular HTML tags, so you can give them both `id` attributes and then look them up with `document.querySelector()`.

The Element objects for both <style> and <link> tags have a disabled property that you can use to disable the entire stylesheet. You might use it with code like this:

```
// This function switches between the "light" and "dark" themes
function toggleTheme() {
    let lightTheme = document.querySelector("#light-theme");
    let darkTheme = document.querySelector("#dark-theme");
    if (darkTheme.disabled) {          // Currently light, switch to dark
        lightTheme.disabled = true;
        darkTheme.disabled = false;
    } else {                           // Currently dark, switch to light
        lightTheme.disabled = false;
        darkTheme.disabled = true;
    }
}
```

Another simple way to script stylesheets is to insert new ones into the document using DOM manipulation techniques we've already seen. For example:

```
function setTheme(name) {
    // Create a new <link rel="stylesheet"> element to load the named stylesheet
    let link = document.createElement("link");
    link.id = "theme";
    link.rel = "stylesheet";
    link.href = `themes/${name}.css`;

    // Look for an existing link with id "theme"
    let currentTheme = document.querySelector("#theme");
    if (currentTheme) {
        // If there is an existing theme, replace it with the new one.
        currentTheme.replaceWith(link);
    } else {
        // Otherwise, just insert the link to the theme stylesheet.
        document.head.append(link);
    }
}
```

Less subtly, you can also just insert a string of HTML containing a <style> tag into your document. This is a fun trick, for example:

```
document.head.insertAdjacentHTML(
    "beforeend",
    "<style>body{transform:rotate(180deg)}</style>"
);
```

Browsers define an API that allows JavaScript to look inside stylesheets to query, modify, insert, and delete style rules in that stylesheet. This API is so specialized that it is not documented here. You can read about it on MDN by searching for "CSSStyleSheet" and "CSS Object Model."

15.4.5 CSS Animations and Events

Suppose you have the following two CSS classes defined in a stylesheet:

```
.transparent { opacity: 0; }
.fadeable { transition: opacity .5s ease-in }
```

If you apply the first style to an element, it will be fully transparent and therefore invisible. But if you apply the second style that tells the browser that when the opacity of the element changes, that change should be animated over a period of 0.5 seconds, "ease-in" specifies that the opacity change animation should start off slow and then accelerate.

Now suppose that your HTML document contains an element with the "fadeable" class:

```
<div id="subscribe" class="fadeable notification">...</div>
```

In JavaScript, you can add the "transparent" class:

```
document.querySelector("#subscribe").classList.add("transparent");
```

This element is configured to animate opacity changes. Adding the "transparent" class changes the opacity and triggers an animate: the browser "fades out" the element so that it becomes fully transparent over the period of half a second.

This works in reverse as well: if you remove the "transparent" class of a "fadeable" element, that is also an opacity change, and the element fades back in and becomes visible again.

JavaScript does not have to do any work to make these animations happen: they are a pure CSS effect. But JavaScript can be used to trigger them.

JavaScript can also be used to monitor the progress of a CSS transition because the web browser fires events at the start and end of a transition. The "transitionrun" event is dispatched when the transition is first triggered. This may happen before any visual changes begin, when the `transition-delay` style has been specified. Once the visual changes begin a "transitionstart" event is dispatched, and when the animation is complete, a "transitionend" event is dispatched. The target of all these events is the element being animated, of course. The event object passed to handlers for these events is a TransitionEvent object. It has a `propertyName` property that specifies the CSS property being animated and an `elapsedTime` property that for "transitionend" events specifies how many seconds have passed since the "transitionstart" event.

In addition to transitions, CSS also supports a more complex form of animation known simply as "CSS Animations." These use CSS properties such as `animation-name` and `animation-duration` and a special `@keyframes` rule to define animation details. Details of how CSS animations work are beyond the scope of this book, but once again, if you define all of the animation properties on a CSS class, then you can

use JavaScript to trigger the animation simply by adding the class to the element that is to be animated.

And like CSS transitions, CSS animations also trigger events that your JavaScript code can listen form. "animationstart" is dispatched when the animation starts, and "animationend" is dispatched when it is complete. If the animation repeats more than once, then an "animationiteration" event is dispatched after each repetition except the last. The event target is the animated element, and the event object passed to handler functions is an AnimationEvent object. These events include an `animationName` property that specifies the `animation-name` property that defines the animation and an `elapsedTime` property that specifies how many seconds have passed since the animation started.

15.5 Document Geometry and Scrolling

In this chapter so far, we have thought about documents as abstract trees of elements and text nodes. But when a browser renders a document within a window, it creates a visual representation of the document in which each element has a position and a size. Often, web applications can treat documents as trees of elements and never have to think about how those elements are rendered on screen. Sometimes, however, it is necessary to determine the precise geometry of an element. If, for example, you want to use CSS to dynamically position an element (such as a tooltip) next to some ordinary browser-positioned element, you need to be able to determine the location of that element.

The following subsections explain how you can go back and forth between the abstract, tree-based *model* of a document and the geometrical, coordinate-based *view* of the document as it is laid out in a browser window.

15.5.1 Document Coordinates and Viewport Coordinates

The position of a document element is measured in CSS pixels, with the *x* coordinate increasing to the right and the *y* coordinate increasing as we go down. There are two different points we can use as the coordinate system origin, however: the *x* and *y* coordinates of an element can be relative to the top-left corner of the document or relative to the top-left corner of the *viewport* in which the document is displayed. In top-level windows and tabs, the "viewport" is the portion of the browser that actually displays document content: it excludes browser "chrome" such as menus, toolbars, and tabs. For documents displayed in `<iframe>` tags, it is the iframe element in the DOM that defines the viewport for the nested document. In either case, when we talk about the position of an element, we must be clear whether we are using document coordinates or viewport coordinates. (Note that viewport coordinates are sometimes called "window coordinates.")

If the document is smaller than the viewport, or if it has not been scrolled, the upper-left corner of the document is in the upper-left corner of the viewport and the document and viewport coordinate systems are the same. In general, however, to convert between the two coordinate systems, we must add or subtract the *scroll offsets*. If an element has a *y* coordinate of 200 pixels in document coordinates, for example, and if the user has scrolled down by 75 pixels, then that element has a *y* coordinate of 125 pixels in viewport coordinates. Similarly, if an element has an *x* coordinate of 400 in viewport coordinates after the user has scrolled the viewport 200 pixels horizontally, then the element's *x* coordinate in document coordinates is 600.

If we use the mental model of printed paper documents, it is logical to assume that every element in a document must have a unique position in document coordinates, regardless of how much the user has scrolled the document. That is an appealing property of paper documents, and it applies for simple web documents, but in general, document coordinates don't really work on the web. The problem is that the CSS `overflow` property allows elements within a document to contain more content than it can display. Elements can have their own scrollbars and serve as viewports for the content they contain. The fact that the web allows scrolling elements within a scrolling document means that it is simply not possible to describe the position of an element within the document using a single (x,y) point.

Because document coordinates don't really work, client-side JavaScript tends to use viewport coordinates. The `getBoundingClientRect()` and `elementFromPoint()` methods described next use viewport coordinates, for example, and the `clientX` and `clientY` properties of mouse and pointer event objects also use this coordinate system.

When you explicitly position an element using CSS `position:fixed`, the `top` and `left` properties are interpreted in viewport coordinates. If you use `position:relative`, the element is positioned relative to where it would have been if it didn't have the `position` property set. If you use `position:absolute`, then `top` and `left` are relative to the document or to the nearest containing positioned element. This means, for example, that an absolutely positioned element inside a relatively positioned element is positioned relative to the container element, not relative to the overall document. It is sometimes very useful to create a relatively positioned container with `top` and `left` set to 0 (so the container is laid out normally) in order to establish a new coordinate system origin for the absolutely positioned elements it contains. We might refer to this new coordinate system as "container coordinates" to distinguish it from document coordinates and viewport coordinates.

CSS Pixels

If, like me, you are old enough to remember computer monitors with resolutions of 1024 × 768 and touch-screen phones with resolutions of 320 × 480, then you may still think that the word "pixel" refers to a single "picture element" in *hardware*. Today's 4K monitors and "retina" displays have such high resolution that software pixels have been decoupled from hardware pixels. A CSS pixel—and therefore a client-side JavaScript pixel—may in fact consist of multiple device pixels. The `devicePixelRatio` property of the Window object specifies how many device pixels are used for each software pixel. A "dpr" of 2, for example, means that each software pixel is actually a 2 × 2 grid of hardware pixels. The `devicePixelRatio` value depends on the physical resolution of your hardware, on settings in your operating system, and on the zoom level in your browser.

`devicePixelRatio` does not have to be an integer. If you are using a CSS font size of "12px" and the device pixel ratio is 2.5, then the actual font size, in device pixels, is 30. Because the pixel values we use in CSS no longer correspond directly to individual pixels on the screen, pixel coordinates no longer need to be integers. If the `devicePixelRatio` is 3, then a coordinate of 3.33 makes perfect sense. And if the ratio is actually 2, then a coordinate of 3.33 will just be rounded up to 3.5.

15.5.2 Querying the Geometry of an Element

You can determine the size (including CSS border and padding, but not the margin) and position (in viewport coordinates) of an element by calling its `getBoundingClientRect()` method. It takes no arguments and returns an object with properties `left`, `right`, `top`, `bottom`, `width`, and `height`. The `left` and `top` properties give the *x* and *y* coordinates of the upper-left corner of the element, and the `right` and `bottom` properties give the coordinates of the lower-right corner. The differences between these values are the `width` and `height` properties.

Block elements, such as images, paragraphs, and `<div>` elements are always rectangular when laid out by the browser. Inline elements, such as `<span>`, `<code>`, and `<b>` elements, however, may span multiple lines and may therefore consist of multiple rectangles. Imagine, for example, some text within `<em>` and `</em>` tags that happens to be displayed so that it wraps across two lines. Its rectangles consist of the end of the first line and beginning of the second line. If you call `getBoundingClientRect()` on this element, the bounding rectangle would include the entire width of both lines. If you want to query the individual rectangles of inline elements, call the `getClientRects()` method to obtain a read-only, array-like object whose elements are rectangle objects like those returned by `getBoundingClientRect()`.

15.5.3 Determining the Element at a Point

The `getBoundingClientRect()` method allows us to determine the current position of an element in a viewport. Sometimes we want to go in the other direction and determine which element is at a given location in the viewport. You can determine this with the `elementFromPoint()` method of the Document object. Call this method with the *x* and *y* coordinates of a point (using viewport coordinates, not document coordinates: the `clientX` and `clientY` coordinates of a mouse event work, for example). `elementFromPoint()` returns an Element object that is at the specified position. The *hit detection* algorithm for selecting the element is not precisely specified, but the intent of this method is that it returns the innermost (most deeply nested) and uppermost (highest CSS `z-index` attribute) element at that point.

15.5.4 Scrolling

The `scrollTo()` method of the Window object takes the *x* and *y* coordinates of a point (in document coordinates) and sets these as the scrollbar offsets. That is, it scrolls the window so that the specified point is in the upper-left corner of the viewport. If you specify a point that is too close to the bottom or too close to the right edge of the document, the browser will move it as close as possible to the upper-left corner but won't be able to get it all the way there. The following code scrolls the browser so that the bottom-most page of the document is visible:

```
// Get the heights of the document and viewport.
let documentHeight = document.documentElement.offsetHeight;
let viewportHeight = window.innerHeight;
// And scroll so the last "page" shows in the viewport
window.scrollTo(0, documentHeight - viewportHeight);
```

The `scrollBy()` method of the Window is similar to `scrollTo()`, but its arguments are relative and are added to the current scroll position:

```
// Scroll 50 pixels down every 500 ms. Note there is no way to turn this off!
setInterval(() => { scrollBy(0,50)}, 500);
```

If you want to scroll smoothly with `scrollTo()` or `scrollBy()`, pass a single object argument instead of two numbers, like this:

```
window.scrollTo({
  left: 0,
  top: documentHeight - viewportHeight,
  behavior: "smooth"
});
```

Often, instead of scrolling to a numeric location in a document, we just want to scroll so that a certain element in the document is visible. You can do this with the `scrollIntoView()` method on the desired HTML element. This method ensures that the element on which it is invoked is visible in the viewport. By default, it tries to put the

top edge of the element at or near the top of the viewport. If `false` is passed as the only argument, it tries to put the bottom edge of the element at the bottom of the viewport. The browser will also scroll the viewport horizontally as needed to make the element visible.

You can also pass an object to `scrollIntoView()`, setting the `behavior:"smooth"` property for smooth scrolling. You can set the `block` property to specify where the element should be positioned vertically and the `inline` property to specify how it should be positioned horizontally if horizontal scrolling is needed. Legal values for both of these properties are `start`, `end`, `nearest`, and `center`.

15.5.5 Viewport Size, Content Size, and Scroll Position

As we've discussed, browser windows and other HTML elements can display scrolling content. When this is the case, we sometimes need to know the size of the viewport, the size of the content, and the scroll offsets of the content within the viewport. This section covers these details.

For browser windows, the viewport size is given by the `window.innerWidth` and `window.innerHeight` properties. (Web pages optimized for mobile devices often use a `<meta name="viewport">` tag in their `<head>` to set the desired viewport width for the page.) The total size of the document is the same as the size of the `<html>` element, `document.documentElement`. You can call `getBoundingClientRect()` on `document.documentElement` to get the width and height of the document, or you can use the `offsetWidth` and `offsetHeight` properties of `document.documentElement`. The scroll offsets of the document within its viewport are available as `window.scrollX` and `window.scrollY`. These are read-only properties, so you can't set them to scroll the document: use `window.scrollTo()` instead.

Things are a little more complicated for elements. Every Element object defines the following three groups of properties:

```
offsetWidth     clientWidth     scrollWidth
offsetHeight    clientHeight    scrollHeight
offsetLeft      clientLeft      scrollLeft
offsetTop       clientTop       scrollTop
offsetParent
```

The `offsetWidth` and `offsetHeight` properties of an element return its on-screen size in CSS pixels. The returned sizes include the element border and padding but not margins. The `offsetLeft` and `offsetTop` properties return the *x* and *y* coordinates of the element. For many elements, these values are document coordinates. But for descendants of positioned elements and for some other elements, such as table cells, these properties return coordinates that are relative to an ancestor element rather

than the document itself. The `offsetParent` property specifies which element the properties are relative to. These offset properties are all read-only.

`clientWidth` and `clientHeight` are like `offsetWidth` and `offsetHeight` except that they do not include the border size—only the content area and its padding. The `clientLeft` and `clientTop` properties are not very useful: they return the horizontal and vertical distance between the outside of an element's padding and the outside of its border. Usually, these values are just the width of the left and top borders. These client properties are all read-only. For inline elements like `<i>`, `<code>`, and `<span>`, they all return 0.

`scrollWidth` and `scrollHeight` return the size of an element's content area plus its padding plus any overflowing content. When the content fits within the content area without overflow, these properties are the same as `clientWidth` and `clientHeight`. But when there is overflow, they include the overflowing content and return values larger than `clientWidth` and `clientHeight`. `scrollLeft` and `scrollTop` give the scroll offset of the element content within the element's viewport. Unlike all the other properties described here, `scrollLeft` and `scrollTop` are writable properties, and you can set them to scroll the content within an element. (In most browsers, Element objects also have `scrollTo()` and `scrollBy()` methods like the Window object does, but these are not yet universally supported.)

15.6 Web Components

HTML is a language for document markup and defines a rich set of tags for that purpose. Over the last three decades, it has become a language that is used to describe the user interfaces of web applications, but basic HTML tags such as `<input>` and `<button>` are inadequate for modern UI designs. Web developers are able to make it work, but only by using CSS and JavaScript to augment the appearance and behavior of basic HTML tags. Consider a typical user interface component, such as the search box shown in Figure 15-3.

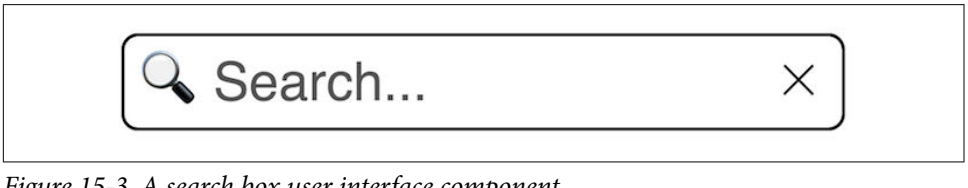

Figure 15-3. A search box user interface component

The HTML `<input>` element can be used to accept a single line of input from the user, but it doesn't have any way to display icons like the magnifying glass on the left and the cancel X on the right. In order to implement a modern user interface element like this for the web, we need to use at least four HTML elements: an `<input>` element

to accept and display the user's input, two `<img>` elements (or in this case, two `<span>` elements displaying Unicode glyphs), and a container `<div>` element to hold those three children. Furthermore, we have to use CSS to hide the default border of the `<input>` element and define a border for the container. And we need to use JavaScript to make all the HTML elements work together. When the user clicks on the X icon, we need an event handler to clear the input from the `<input>` element, for example.

That is a lot of work to do every time you want to display a search box in a web application, and most web applications today are not written using "raw" HTML. Instead, many web developers use frameworks like React and Angular that support the creation of reusable user interface components like the search box shown here. Web components is a browser-native alternative to those frameworks based on three relatively recent additions to web standards that allow JavaScript to extend HTML with new tags that work as self-contained, reusable UI components.

The subsections that follow explain how to use web components defined by other developers in your own web pages, then explain each of the three technologies that web components are based on, and finally tie all three together in an example that implements the search box element pictured in Figure 15-3.

15.6.1 Using Web Components

Web components are defined in JavaScript, so in order to use a web component in your HTML file, you need to include the JavaScript file that defines the component. Because web components are a relatively new technology, they are often written as JavaScript modules, so you might include one in your HTML like this:

```
<script type="module" src="components/search-box.js">
```

Web components define their own HTML tag names, with the important restriction that those tag names must include a hyphen. (This means that future versions of HTML can introduce new tags without hyphens, and there is no chance that the tags will conflict with anyone's web component.) To use a web component, just use its tag in your HTML file:

```
<search-box placeholder="Search..."></search-box>
```

Web components can have attributes just like regular HTML tags can; the documentation for the component you are using should tell you which attributes are supported. Web components cannot be defined with self-closing tags. You cannot write `<search-box/>`, for example. Your HTML file must include both the opening tag and the closing tag.

Like regular HTML elements, some web components are written to expect children and others are written in such a way that they do not expect (and will not display) children. Some web components are written so that they can optionally accept

specially labeled children that will appear in named "slots." The `<search-box>` component pictured in Figure 15-3 and implemented in Example 15-3 uses "slots" for the two icons it displays. If you want to to use a `<search-box>` with different icons, you can use HTML like this:

```
<search-box>
  <img src="images/search-icon.png" slot="left"/>
  <img src="images/cancel-icon.png" slot="right"/>
</search-box>
```

The `slot` attribute is an extension to HTML that it is used to specify which children should go where. The slot names—"left" and "right" in this example—are defined by the web component. If the component you are using supports slots, that fact should be included in its documentation.

I previously noted that web components are often implemented as JavaScript modules and can be loaded into HTML files with a `<script type="module">` tag. You may remember from the beginning of this chapter that modules are loaded after document content is parsed, as if they had a `deferred` tag. So this means that a web browser will typically parse and render tags like `<search-box>` before it has run the code that will tell it what a `<search-box>` is. This is normal when using web components. HTML parsers in web browsers are flexible and very forgiving about input that they do not understand. When they encounter a web component tag before that component has been defined, they add a generic HTMLElement to the DOM tree even though they do not know what to do with it. Later, when the custom element is defined, the generic element is "upgraded" so that it looks and behaves as desired.

If a web component has children, then those children will probably be displayed incorrectly before the component is defined. You can use this CSS to keep web components hidden until they are defined:

```
/*
 * Make the <search-box> component invisible before it is defined.
 * And try to duplicate its eventual layout and size so that nearby
 * content does not move when it becomes defined.
 */
search-box:not(:defined) {
    opacity:0;
    display: inline-block;
    width: 300px;
    height: 50px;
}
```

Like regular HTML elements, web components can be used in JavaScript. If you include a `<search-box>` tag in your web page, then you can obtain a reference to it with `querySelector()` and an appropriate CSS selector, just as you would for any other HTML tag. Generally, it only makes sense to do this after the module that defines the component has run, so be careful when querying web components that

you do not do so too early. Web component implementations typically (but this is not a requirement) define a JavaScript property for each HTML attribute they support. And, like HTML elements, they may also define useful methods. Once again, the documentation for the web component you are using should specify what properties and methods are available to your JavaScript code.

Now that you know how to use web components, the next three sections cover the three web browser features that allow us to implement them.

DocumentFragment Nodes

Before we can cover web component APIs, we need to return briefly to the DOM API to explain what a DocumentFragment is. The DOM API organizes a document into a tree of Node objects, where a Node can be a Document, an Element, a Text node, or even a Comment. None of these node types allows you to represent a fragment of a document that consists of a set of sibling nodes without their parent. This is where DocumentFragment comes in: it is another type of Node that serves as a temporary parent when you want to manipulate a group of sibling nodes as a single unit. You can create a DocumentFragment node with `document.createDocumentFragment()`. Once you have a DocumentFragment, you can use it like an Element and `append()` content to it. A DocumentFragment is different from an Element because it does not have a parent. But more importantly, when you insert a DocumentFragment node into the document, the DocumentFragment itself is not inserted. Instead, all of its children are inserted.

15.6.2 HTML Templates

The HTML `<template>` tag is only loosely related to web components, but it does enable a useful optimization for components that appear frequently in web pages. `<template>` tags and their children are never rendered by a web browser and are only useful on web pages that use JavaScript. The idea behind this tag is that when a web page contains multiple repetitions of the same basic HTML structure (such as rows in a table or the internal implementation of a web component), then we can use a `<template>` to define that element structure once, then use JavaScript to duplicate the structure as many times as needed.

In JavaScript, a `<template>` tag is represented by an HTMLTemplateElement object. This object defines a single `content` property, and the value of this property is a DocumentFragment of all the child nodes of the `<template>`. You can clone this DocumentFragment and then insert the cloned copy into your document as needed. The fragment itself will not be inserted, but its children will be. Suppose you're working with a document that includes a `<table>` and `<template id="row">` tag and that the

template defines the structure of rows for that table. You might use the template like this:

```
let tableBody = document.querySelector("tbody");
let template = document.querySelector("#row");
let clone = template.content.cloneNode(true);  // deep clone
// ...Use the DOM to insert content into the <td> elements of the clone...
// Now add the cloned and initialized row into the table
tableBody.append(clone);
```

Template elements do not have to appear literally in an HTML document in order to be useful. You can create a template in your JavaScript code, create its children with `innerHTML`, and then make as many clones as needed without the parsing overhead of `innerHTML`. This is how HTML templates are typically used in web components, and Example 15-3 demonstrates this technique.

15.6.3 Custom Elements

The second web browser feature that enables web components is "custom elements": the ability to associate a JavaScript class with an HTML tag name so that any such tags in the document are automatically turned into instances of the class in the DOM tree. The `customElements.define()` method takes a web component tag name as its first argument (remember that the tag name must include a hyphen) and a subclass of HTMLElement as its second argument. Any existing elements in the document with that tag name are "upgraded" to newly created instances of the class. And if the browser parses any HTML in the future, it will automatically create an instance of the class for each of the tags it encounters.

The class passed to `customElements.define()` should extend HTMLElement and not a more specific type like HTMLButtonElement.[4] Recall from Chapter 9 that when a JavaScript class extends another class, the constructor function must call `super()` before it uses the `this` keyword, so if the custom element class has a constructor, it should call `super()` (with no arguments) before doing anything else.

The browser will automatically invoke certain "lifecycle methods" of a custom element class. The `connectedCallback()` method is invoked when an instance of the custom element is inserted into the document, and many elements use this method to perform initialization. There is also a `disconnectedCallback()` method invoked when (and if) the element is removed from the document, though this is less often used.

4 The custom element specification allows subclassing of `<button>` and other specific element classes, but this is not supported in Safari and a different syntax is required to use a custom element that extends anything other than HTMLElement.

If a custom element class defines a static `observedAttributes` property whose value is an array of attribute names, and if any of the named attributes are set (or changed) on an instance of the custom element, the browser will invoke the `attributeChangedCallback()` method, passing the attribute name, its old value, and its new value. This callback can take whatever steps are necessary to update the component based on its attribute values.

Custom element classes can also define whatever other properties and methods they want to. Commonly, they will define getter and setter methods that make the element's attributes available as JavaScript properties.

As an example of a custom element, suppose we want to be able to display circles within paragraphs of regular text. We'd like to be able to write HTML like this in order to render mathematical story problems like the one shown in Figure 15-4:

```
<p>
  The document has one marble: <inline-circle></inline-circle>.
  The HTML parser instantiates two more marbles:
  <inline-circle diameter="1.2em" color="blue"></inline-circle>
  <inline-circle diameter=".6em" color="gold"></inline-circle>.
  How many marbles does the document contain now?
</p>
```

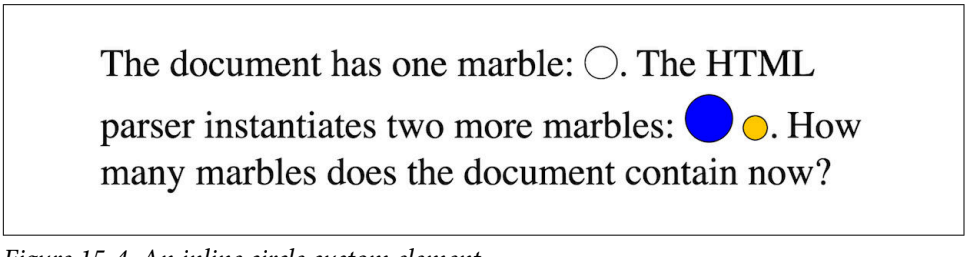

Figure 15-4. An inline circle custom element

We can implement this `<inline-circle>` custom element with the code shown in Example 15-2:

Example 15-2. The <inline-circle> custom element

```
customElements.define("inline-circle", class InlineCircle extends HTMLElement {
    // The browser calls this method when an <inline-circle> element
    // is inserted into the document. There is also a disconnectedCallback()
    // that we don't need in this example.
    connectedCallback() {
        // Set the styles needed to create circles
        this.style.display = "inline-block";
        this.style.borderRadius = "50%";
        this.style.border = "solid black 1px";
        this.style.transform = "translateY(10%)";
```

```
        // If there is not already a size defined, set a default size
        // that is based on the current font size.
        if (!this.style.width) {
            this.style.width = "0.8em";
            this.style.height = "0.8em";
        }
    }

    // The static observedAttributes property specifies which attributes
    // we want to be notified about changes to. (We use a getter here since
    // we can only use "static" with methods.)
    static get observedAttributes() { return ["diameter", "color"]; }

    // This callback is invoked when one of the attributes listed above
    // changes, either when the custom element is first parsed, or later.
    attributeChangedCallback(name, oldValue, newValue) {
        switch(name) {
        case "diameter":
            // If the diameter attribute changes, update the size styles
            this.style.width = newValue;
            this.style.height = newValue;
            break;
        case "color":
            // If the color attribute changes, update the color styles
            this.style.backgroundColor = newValue;
            break;
        }
    }

    // Define JavaScript properties that correspond to the element's
    // attributes. These getters and setters just get and set the underlying
    // attributes. If a JavaScript property is set, that sets the attribute
    // which triggers a call to attributeChangedCallback() which updates
    // the element styles.
    get diameter() { return this.getAttribute("diameter"); }
    set diameter(diameter) { this.setAttribute("diameter", diameter); }
    get color() { return this.getAttribute("color"); }
    set color(color) { this.setAttribute("color", color); }
});
```

15.6.4 Shadow DOM

The custom element demonstrated in Example 15-2 is not well encapsulated. When you set its `diameter` or `color` attributes, it responds by altering its own `style` attribute, which is not behavior we would ever expect from a real HTML element. To turn a custom element into a true web component, it should use the powerful encapsulation mechanism known as *shadow DOM*.

Shadow DOM allows a "shadow root" to be attached to a custom element (and also to a `<div>`, `<span>`, `<body>`, `<article>`, `<main>`, `<nav>`, `<header>`, `<footer>`, `<section>`,

<p>, <blockquote>, <aside>, or <h1> through <h6> element) known as a "shadow host." Shadow host elements, like all HTML elements, are already the root of a normal DOM tree of descendant elements and text nodes. A shadow root is the root of another, more private, tree of descendant elements that sprouts from the shadow host and can be thought of as a distinct minidocument.

The word "shadow" in "shadow DOM" refers to the fact that elements that descend from a shadow root are "hiding in the shadows": they are not part of the normal DOM tree, do not appear in the `children` array of their host element, and are not visited by normal DOM traversal methods such as `querySelector()`. For contrast, the normal, regular DOM children of a shadow host are sometimes referred to as the "light DOM."

To understand the purpose of the shadow DOM, picture the HTML <audio> and <video> elements: they display a nontrivial user interface for controlling media playback, but the play and pause buttons and other UI elements are not part of the DOM tree and cannot be manipulated by JavaScript. Given that web browsers are designed to display HTML, it is only natural that browser vendors would want to display internal UIs like these using HTML. In fact, most browsers have been doing something like that for a long time, and the shadow DOM makes it a standard part of the web platform.

Shadow DOM encapsulation

The key feature of shadow DOM is the encapsulation it provides. The descendants of a shadow root are hidden from—and independent from—the regular DOM tree, almost as if they were in an independent document. There are three very important kinds of encapsulation provided by the shadow DOM:

- As already mentioned, elements in the shadow DOM are hidden from regular DOM methods like `querySelectorAll()`. When a shadow root is created and attached to its shadow host, it can be created in "open" or "closed" mode. A closed shadow root is completely sealed away and inaccessible. More commonly, though, shadow roots are created in "open" mode, which means that the shadow host has a `shadowRoot` property that JavaScript can use to gain access to the elements of the shadow root, if it has some reason to do so.
- Styles defined beneath a shadow root are private to that tree and will never affect the light DOM elements on the outside. (A shadow root can define default styles for its host element, but these will be overridden by light DOM styles.) Similarly, the light DOM styles that apply to the shadow host element have no effect on the descendants of the shadow root. Elements in the shadow DOM will inherit things like font size and background color from the light DOM, and styles in the shadow DOM can choose to use CSS variables defined in the light DOM. For the most part, however, the styles of the light DOM and the styles of the shadow

DOM are completely independent: the author of a web component and the user of a web component do not have to worry about collisions or conflicts between their stylesheets. Being able to "scope" CSS in this way is perhaps the most important feature of the shadow DOM.

- Some events (like "load") that occur within the shadow DOM are confined to the shadow DOM. Others, including focus, mouse, and keyboard events bubble up and out. When an event that originates in the shadow DOM crosses the boundary and begins to propagate in the light DOM, its `target` property is changed to the shadow host element, so it appears to have originated directly on that element.

Shadow DOM slots and light DOM children

An HTML element that is a shadow host has two trees of descendants. One is the `children[]` array—the regular light DOM descendants of the host element—and the other is the shadow root and all of its descendants, and you may be wondering how two distinct content trees can be displayed within the same host element. Here's how it works:

- The descendants of the shadow root are always displayed within the shadow host.
- If those descendants include a `<slot>` element, then the regular light DOM children of the host element are displayed as if they were children of that `<slot>`, replacing any shadow DOM content in the slot. If the shadow DOM does not include a `<slot>`, then any light DOM content of the host is never displayed. If the shadow DOM has a `<slot>`, but the shadow host has no light DOM children, then the shadow DOM content of the slot is displayed as a default.
- When light DOM content is displayed within a shadow DOM slot, we say that those elements have been "distributed," but it is important to understand that the elements do not actually become part of the shadow DOM. They can still be queried with `querySelector()`, and they still appear in the light DOM as children or descendants of the host element.
- If the shadow DOM defines more than one `<slot>` and names those slots with a `name` attribute, then children of the shadow host can specify which slot they would like to appear in by specifying a `slot="slotname"` attribute. We saw an example of this usage in §15.6.1 when we demonstrated how to customize the icons displayed by the `<search-box>` component.

Shadow DOM API

For all of its power, the Shadow DOM doesn't have much of a JavaScript API. To turn a light DOM element into a shadow host, just call its `attachShadow()` method, passing `{mode:"open"}` as the only argument. This method returns a shadow root object and also sets that object as the value of the host's `shadowRoot` property. The shadow root object is a DocumentFragment, and you can use DOM methods to add content to it or just set its `innerHTML` property to a string of HTML.

If your web component needs to know when the light DOM content of a shadow DOM `<slot>` has changed, it can register a listener for "slotchanged" events directly on the `<slot>` element.

15.6.5 Example: a <search-box> Web Component

Figure 15-3 illustrated a `<search-box>` web component. Example 15-3 demonstrates the three enabling technologies that define web components: it implements the `<search-box>` component as a custom element that uses a `<template>` tag for efficiency and a shadow root for encapsulation.

This example shows how to use the low-level web component APIs directly. In practice, many web components developed today create them using higher-level libraries such as "lit-element." One of the reasons to use a library is that creating reusable and customizable components is actually quite hard to do well, and there are many details to get right. Example 15-3 demonstrates web components and does some basic keyboard focus handling, but otherwise ignores accessibility and makes no attempt to use proper ARIA attributes to make the component work with screen readers and other assistive technology.

Example 15-3. Implementing a web component

```
/**
 * This class defines a custom HTML <search-box> element that displays an
 * <input> text input field plus two icons or emoji. By default, it displays a
 * magnifying glass emoji (indicating search) to the left of the text field
 * and an X emoji (indicating cancel) to the right of the text field. It
 * hides the border on the input field and displays a border around itself,
 * creating the appearance that the two emoji are inside the input
 * field. Similarly, when the internal input field is focused, the focus ring
 * is displayed around the <search-box>.
 *
 * You can override the default icons by including <span> or <img> children
 * of <search-box> with slot="left" and slot="right" attributes.
 *
 * <search-box> supports the normal HTML disabled and hidden attributes and
 * also size and placeholder attributes, which have the same meaning for this
 * element as they do for the <input> element.
```

```
 *
 * Input events from the internal <input> element bubble up and appear with
 * their target field set to the <search-box> element.
 *
 * The element fires a "search" event with the detail property set to the
 * current input string when the user clicks on the left emoji (the magnifying
 * glass). The "search" event is also dispatched when the internal text field
 * generates a "change" event (when the text has changed and the user types
 * Return or Tab).
 *
 * The element fires a "clear" event when the user clicks on the right emoji
 * (the X). If no handler calls preventDefault() on the event then the element
 * clears the user's input once event dispatch is complete.
 *
 * Note that there are no onsearch and onclear properties or attributes:
 * handlers for the "search" and "clear" events can only be registered with
 * addEventListener().
 */
class SearchBox extends HTMLElement {
    constructor() {
        super(); // Invoke the superclass constructor; must be first.

        // Create a shadow DOM tree and attach it to this element, setting
        // the value of this.shadowRoot.
        this.attachShadow({mode: "open"});

        // Clone the template that defines the descendants and stylesheet for
        // this custom component, and append that content to the shadow root.
        this.shadowRoot.append(SearchBox.template.content.cloneNode(true));

        // Get references to the important elements in the shadow DOM
        this.input = this.shadowRoot.querySelector("#input");
        let leftSlot = this.shadowRoot.querySelector('slot[name="left"]');
        let rightSlot = this.shadowRoot.querySelector('slot[name="right"]');

        // When the internal input field gets or loses focus, set or remove
        // the "focused" attribute which will cause our internal stylesheet
        // to display or hide a fake focus ring on the entire component. Note
        // that the "blur" and "focus" events bubble and appear to originate
        // from the <search-box>.
        this.input.onfocus = () => { this.setAttribute("focused", ""); };
        this.input.onblur = () => { this.removeAttribute("focused");};

        // If the user clicks on the magnifying glass, trigger a "search"
        // event.  Also trigger it if the input field fires a "change"
        // event. (The "change" event does not bubble out of the Shadow DOM.)
        leftSlot.onclick = this.input.onchange = (event) => {
            event.stopPropagation();    // Prevent click events from bubbling
            if (this.disabled) return;  // Do nothing when disabled
            this.dispatchEvent(new CustomEvent("search", {
                detail: this.input.value
            }));
```

```
    };

    // If the user clicks on the X, trigger a "clear" event.
    // If preventDefault() is not called on the event, clear the input.
    rightSlot.onclick = (event) => {
        event.stopPropagation();    // Don't let the click bubble up
        if (this.disabled) return;  // Don't do anything if disabled
        let e = new CustomEvent("clear", { cancelable: true });
        this.dispatchEvent(e);
        if (!e.defaultPrevented) {  // If the event was not "cancelled"
            this.input.value = "";  // then clear the input field
        }
    };
}

// When some of our attributes are set or changed, we need to set the
// corresponding value on the internal <input> element. This life cycle
// method, together with the static observedAttributes property below,
// takes care of that.
attributeChangedCallback(name, oldValue, newValue) {
    if (name === "disabled") {
        this.input.disabled = newValue !== null;
    } else if (name === "placeholder") {
        this.input.placeholder = newValue;
    } else if (name === "size") {
        this.input.size = newValue;
    } else if (name === "value") {
        this.input.value = newValue;
    }
}

// Finally, we define property getters and setters for properties that
// correspond to the HTML attributes we support. The getters simply return
// the value (or the presence) of the attribute. And the setters just set
// the value (or the presence) of the attribute. When a setter method
// changes an attribute, the browser will automatically invoke the
// attributeChangedCallback above.

get placeholder() { return this.getAttribute("placeholder"); }
get size() { return this.getAttribute("size"); }
get value() { return this.getAttribute("value"); }
get disabled() { return this.hasAttribute("disabled"); }
get hidden() { return this.hasAttribute("hidden"); }

set placeholder(value) { this.setAttribute("placeholder", value); }
set size(value) { this.setAttribute("size", value); }
set value(text) { this.setAttribute("value", text); }
set disabled(value) {
    if (value) this.setAttribute("disabled", "");
    else this.removeAttribute("disabled");
}
set hidden(value) {
```

```
        if (value) this.setAttribute("hidden", "");
        else this.removeAttribute("hidden");
    }
}

// This static field is required for the attributeChangedCallback method.
// Only attributes named in this array will trigger calls to that method.
SearchBox.observedAttributes = ["disabled", "placeholder", "size", "value"];

// Create a <template> element to hold the stylesheet and the tree of
// elements that we'll use for each instance of the SearchBox element.
SearchBox.template = document.createElement("template");

// We initialize the template by parsing this string of HTML. Note, however,
// that when we instantiate a SearchBox, we are able to just clone the nodes
// in the template and do have to parse the HTML again.
SearchBox.template.innerHTML = `
<style>
/*
 * The :host selector refers to the <search-box> element in the light
 * DOM. These styles are defaults and can be overridden by the user of the
 * <search-box> with styles in the light DOM.
 */
:host {
  display: inline-block;   /* The default is inline display */
  border: solid black 1px; /* A rounded border around the <input> and <slots> */
  border-radius: 5px;
  padding: 4px 6px;        /* And some space inside the border */
}
:host([hidden]) {          /* Note the parentheses: when host has hidden... */
  display:none;            /* ...attribute set don't display it */
}
:host([disabled]) {        /* When host has the disabled attribute... */
  opacity: 0.5;            /* ...gray it out */
}
:host([focused]) {         /* When host has the focused attribute... */
  box-shadow: 0 0 2px 2px #6AE;  /* display this fake focus ring. */
}

/* The rest of the stylesheet only applies to elements in the Shadow DOM. */
input {
  border-width: 0;         /* Hide the border of the internal input field. */
  outline: none;           /* Hide the focus ring, too. */
  font: inherit;           /* <input> elements don't inherit font by default */
  background: inherit;     /* Same for background color. */
}
slot {
  cursor: default;         /* An arrow pointer cursor over the buttons */
  user-select: none;       /* Don't let the user select the emoji text */
}
</style>
<div>
```

```
  <slot name="left">\u{1f50d}</slot>  <!-- U+1F50D is a magnifying glass -->
  <input type="text" id="input" />    <!-- The actual input element -->
  <slot name="right">\u{2573}</slot>  <!-- U+2573 is an X -->
</div>
`;

// Finally, we call customElement.define() to register the SearchBox element
// as the implementation of the <search-box> tag. Custom elements are required
// to have a tag name that contains a hyphen.
customElements.define("search-box", SearchBox);
```

15.7 SVG: Scalable Vector Graphics

SVG (scalable vector graphics) is an image format. The word "vector" in its name indicates that it is fundamentally different from raster image formats, such as GIF, JPEG, and PNG, that specify a matrix of pixel values. Instead, an SVG "image" is a precise, resolution-independent (hence "scalable") description of the steps necessary to draw the desired graphic. SVG images are described by text files using the XML markup language, which is quite similar to HTML.

There are three ways you can use SVG in web browsers:

1. You can use *.svg* image files with regular HTML `<img>` tags, just as you would use a *.png* or *.jpeg* image.
2. Because the XML-based SVG format is so similar to HTML, you can actually embed SVG tags directly into your HTML documents. If you do this, the browser's HTML parser allows you to omit XML namespaces and treat SVG tags as if they were HTML tags.
3. You can use the DOM API to dynamically create SVG elements to generate images on demand.

The subsections that follow demonstrate the second and third uses of SVG. Note, however, that SVG has a large and moderately complex grammar. In addition to simple shape-drawing primitives, it includes support for arbitrary curves, text, and animation. SVG graphics can even incorporate JavaScript scripts and CSS stylesheets to add behavior and presentation information. A full description of SVG is well beyond the scope of this book. The goal of this section is just to show you how you can use SVG in your HTML documents and script it with JavaScript.

15.7.1 SVG in HTML

SVG images can, of course, be displayed using HTML `<img>` tags. But you can also embed SVG directly in HTML. And if you do this, you can even use CSS stylesheets to specify things like fonts, colors, and line widths. Here, for example, is an HTML file that uses SVG to display an analog clock face:

```
<html>
<head>
<title>Analog Clock</title>
<style>
/* These CSS styles all apply to the SVG elements defined below */
#clock {                              /* Styles for everything in the clock:*/
   stroke: black;                     /* black lines */
   stroke-linecap: round;             /* with rounded ends */
   fill: #ffe;                        /* on an off-white background */
}
#clock .face { stroke-width: 3; }     /* Clock face outline */
#clock .ticks { stroke-width: 2; }    /* Lines that mark each hour */
#clock .hands { stroke-width: 3; }    /* How to draw the clock hands */
#clock .numbers {                     /* How to draw the numbers */
    font-family: sans-serif; font-size: 10; font-weight: bold;
    text-anchor: middle; stroke: none; fill: black;
}
</style>
</head>
<body>
  <svg id="clock" viewBox="0 0 100 100" width="250" height="250">
    <!-- The width and height attributes are the screen size of the graphic -->
    <!-- The viewBox attribute gives the internal coordinate system -->
    <circle class="face" cx="50" cy="50" r="45"/>  <!-- the clock face -->
    <g class="ticks">   <!-- tick marks for each of the 12 hours -->
      <line x1='50' y1='5.000' x2='50.00' y2='10.00'/>
      <line x1='72.50' y1='11.03' x2='70.00' y2='15.36'/>
      <line x1='88.97' y1='27.50' x2='84.64' y2='30.00'/>
      <line x1='95.00' y1='50.00' x2='90.00' y2='50.00'/>
      <line x1='88.97' y1='72.50' x2='84.64' y2='70.00'/>
      <line x1='72.50' y1='88.97' x2='70.00' y2='84.64'/>
      <line x1='50.00' y1='95.00' x2='50.00' y2='90.00'/>
      <line x1='27.50' y1='88.97' x2='30.00' y2='84.64'/>
      <line x1='11.03' y1='72.50' x2='15.36' y2='70.00'/>
      <line x1='5.000' y1='50.00' x2='10.00' y2='50.00'/>
      <line x1='11.03' y1='27.50' x2='15.36' y2='30.00'/>
      <line x1='27.50' y1='11.03' x2='30.00' y2='15.36'/>
    </g>
    <g class="numbers"> <!-- Number the cardinal directions-->
      <text x="50" y="18">12</text><text x="85" y="53">3</text>
      <text x="50" y="88">6</text><text x="15" y="53">9</text>
    </g>
    <g class="hands">   <!-- Draw hands pointing straight up. -->
      <line class="hourhand" x1="50" y1="50" x2="50" y2="25"/>
      <line class="minutehand" x1="50" y1="50" x2="50" y2="20"/>
    </g>
  </svg>
  <script src="clock.js"></script>
</body>
</html>
```

You'll notice that the descendants of the `<svg>` tag are not normal HTML tags. `<circle>`, `<line>`, and `<text>` tags have obvious purposes, though, and it should be clear how this SVG graphic works. There are many other SVG tags, however, and you'll need to consult an SVG reference to learn more. You may also notice that the stylesheet is odd. Styles like `fill`, `stroke-width`, and `text-anchor` are not normal CSS style properties. In this case, CSS is essentially being used to set attributes of SVG tags that appear in the document. Note also that the CSS `font` shorthand property does not work for SVG tags, and you must explicitly set `font-family`, `font-size`, and `font-weight` as separate style properties.

15.7.2 Scripting SVG

One reason to embed SVG directly into your HTML files (instead of just using static `<img>` tags) is that if you do this, then you can use the DOM API to manipulate the SVG image. Suppose you use SVG to display icons in your web application. You could embed SVG within a `<template>` tag (§15.6.2) and then clone the template content whenever you need to insert a copy of that icon into your UI. And if you want the icon to respond to user activity—by changing color when the user hovers the pointer over it, for example—you can often achieve this with CSS.

It is also possible to dynamically manipulate SVG graphics that are directly embedded in HTML. The clock face example in the previous section displays a static clock with hour and minute hands facing straight up displaying the time noon or midnight. But you may have noticed that the HTML file includes a `<script>` tag. That script runs a function periodically to check the time and transform the hour and minute hands by rotating them the appropriate number of degrees so that the clock actually displays the current time, as shown in Figure 15-5.

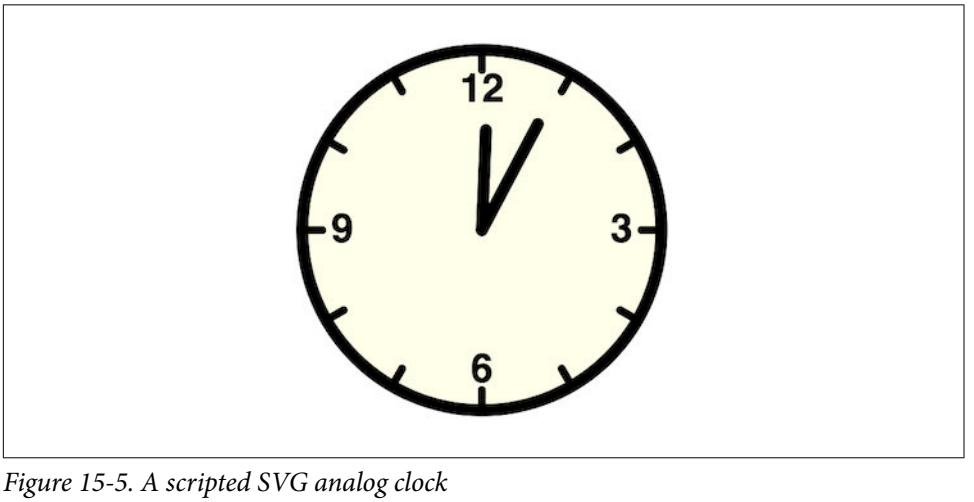

Figure 15-5. A scripted SVG analog clock

The code to manipulate the clock is straightforward. It determines the proper angle of the hour and minute hands based on the current time, then uses `querySelector()` to look up the SVG elements that display those hands, then sets a `transform` attribute on them to rotate them around the center of the clock face. The function uses `setTimeout()` to ensure that it runs once a minute:

```
(function updateClock() { // Update the SVG clock graphic to show current time
    let now = new Date();                       // Current time
    let sec = now.getSeconds();                 // Seconds
    let min = now.getMinutes() + sec/60;        // Fractional minutes
    let hour = (now.getHours() % 12) + min/60;  // Fractional hours
    let minangle = min * 6;                     // 6 degrees per minute
    let hourangle = hour * 30;                  // 30 degrees per hour

    // Get SVG elements for the hands of the clock
    let minhand = document.querySelector("#clock .minutehand");
    let hourhand = document.querySelector("#clock .hourhand");

    // Set an SVG attribute on them to move them around the clock face
    minhand.setAttribute("transform", `rotate(${minangle},50,50)`);
    hourhand.setAttribute("transform", `rotate(${hourangle},50,50)`);

    // Run this function again in 10 seconds
    setTimeout(updateClock, 10000);
}()); // Note immediate invocation of the function here.
```

15.7.3 Creating SVG Images with JavaScript

In addition to simply scripting SVG images embedded in your HTML documents, you can also build SVG images from scratch, which can be useful to create visualizations of dynamically loaded data, for example. Example 15-4 demonstrates how you can use JavaScript to create SVG pie charts, like the one shown in Figure 15-6.

Even though SVG tags can be included within HTML documents, they are technically XML tags, not HTML tags, and if you want to create SVG elements with the JavaScript DOM API, you can't use the normal `createElement()` function that was introduced in §15.3.5. Instead you must use `createElementNS()`, which takes an XML namespace string as its first argument. For SVG, that namespace is the literal string "http://www.w3.org/2000/svg."

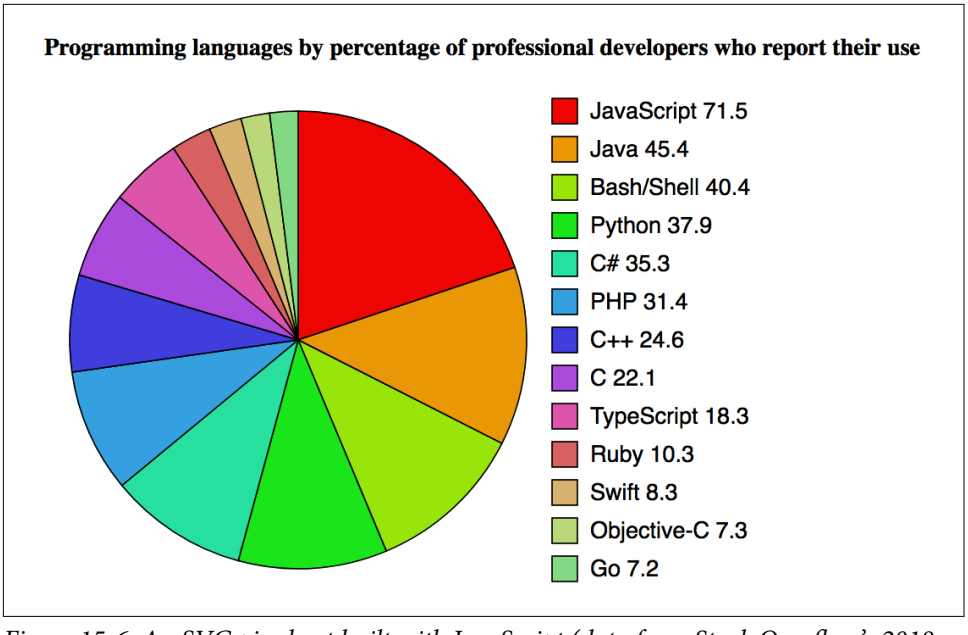

Figure 15-6. An SVG pie chart built with JavaScript (data from Stack Overflow's 2018 Developer Survey of Most Popular Technologies)

Other than the use of `createElementNS()`, the pie chart–drawing code in Example 15-4 is relatively straightforward. There is a little math to convert the data being charted into pie-slice angles. The bulk of the example, however, is DOM code that creates SVG elements and sets attributes on those elements.

The most opaque part of this example is the code that draws the actual pie slices. The element used to display each slice is `<path>`. This SVG element describes arbitrary shapes comprised of lines and curves. The shape description is specified by the `d` attribute of the `<path>` element. The value of this attribute uses a compact grammar of letter codes and numbers that specify coordinates, angles, and other values. The letter M, for example, means "move to" and is followed by *x* and *y* coordinates. The letter L means "line to" and draws a line from the current point to the coordinates that follow it. This example also uses the letter A to draw an arc. This letter is followed by seven numbers describing the arc, and you can look up the syntax online if you want to know more.

Example 15-4. Drawing a pie chart with JavaScript and SVG

```
/**
 * Create an <svg> element and draw a pie chart into it.
 *
 * This function expects an object argument with the following properties:
 *
 *   width, height: the size of the SVG graphic, in pixels
 *   cx, cy, r: the center and radius of the pie
 *   lx, ly: the upper-left corner of the chart legend
 *   data: an object whose property names are data labels and whose
 *         property values are the values associated with each label
 *
 * The function returns an <svg> element. The caller must insert it into
 * the document in order to make it visible.
 */
function pieChart(options) {
    let {width, height, cx, cy, r, lx, ly, data} = options;

    // This is the XML namespace for svg elements
    let svg = "http://www.w3.org/2000/svg";

    // Create the <svg> element, and specify pixel size and user coordinates
    let chart = document.createElementNS(svg, "svg");
    chart.setAttribute("width", width);
    chart.setAttribute("height", height);
    chart.setAttribute("viewBox", `0 0 ${width} ${height}`);

    // Define the text styles we'll use for the chart. If we leave these
    // values unset here, they can be set with CSS instead.
    chart.setAttribute("font-family", "sans-serif");
    chart.setAttribute("font-size", "18");

    // Get labels and values as arrays and add up the values so we know how
    // big the pie is.
    let labels = Object.keys(data);
    let values = Object.values(data);
    let total = values.reduce((x,y) => x+y);

    // Figure out the angles for all the slices. Slice i starts at angles[i]
    // and ends at angles[i+1]. The angles are measured in radians.
    let angles = [0];
    values.forEach((x, i) => angles.push(angles[i] + x/total * 2 * Math.PI));

    // Now loop through the slices of the pie
    values.forEach((value, i) => {
        // Compute the two points where our slice intersects the circle
        // These formulas are chosen so that an angle of 0 is at 12 o'clock
        // and positive angles increase clockwise.
        let x1 = cx + r * Math.sin(angles[i]);
        let y1 = cy - r * Math.cos(angles[i]);
        let x2 = cx + r * Math.sin(angles[i+1]);
```

```
        let y2 = cy - r * Math.cos(angles[i+1]);

        // This is a flag for angles larger than a half circle
        // It is required by the SVG arc drawing component
        let big = (angles[i+1] - angles[i] > Math.PI) ? 1 : 0;

        // This string describes how to draw a slice of the pie chart:
        let path = `M${cx},${cy}` +     // Move to circle center.
            `L${x1},${y1}` +            // Draw line to (x1,y1).
            `A${r},${r} 0 ${big} 1` +   // Draw an arc of radius r...
            `${x2},${y2}` +             // ...ending at to (x2,y2).
            "Z";                        // Close path back to (cx,cy).

        // Compute the CSS color for this slice. This formula works for only
        // about 15 colors. So don't include more than 15 slices in a chart.
        let color = `hsl(${(i*40)%360},${90-3*i}%,${50+2*i}%)`;

        // We describe a slice with a <path> element. Note createElementNS().
        let slice = document.createElementNS(svg, "path");

        // Now set attributes on the <path> element
        slice.setAttribute("d", path);           // Set the path for this slice
        slice.setAttribute("fill", color);       // Set slice color
        slice.setAttribute("stroke", "black");   // Outline slice in black
        slice.setAttribute("stroke-width", "1"); // 1 CSS pixel thick
        chart.append(slice);                     // Add slice to chart

        // Now draw a little matching square for the key
        let icon = document.createElementNS(svg, "rect");
        icon.setAttribute("x", lx);              // Position the square
        icon.setAttribute("y", ly + 30*i);
        icon.setAttribute("width", 20);          // Size the square
        icon.setAttribute("height", 20);
        icon.setAttribute("fill", color);        // Same fill color as slice
        icon.setAttribute("stroke", "black");    // Same outline, too.
        icon.setAttribute("stroke-width", "1");
        chart.append(icon);                      // Add to the chart

        // And add a label to the right of the rectangle
        let label = document.createElementNS(svg, "text");
        label.setAttribute("x", lx + 30);        // Position the text
        label.setAttribute("y", ly + 30*i + 16);
        label.append(`${labels[i]} ${value}`);   // Add text to label
        chart.append(label);                     // Add label to the chart
    });

    return chart;
}
```

The pie chart in Figure 15-6 was created using the `pieChart()` function from Example 15-4, like this:

```
document.querySelector("#chart").append(pieChart({
    width: 640, height:400,    // Total size of the chart
    cx: 200, cy: 200, r: 180,  // Center and radius of the pie
    lx: 400, ly: 10,           // Position of the legend
    data: {                    // The data to chart
        "JavaScript": 71.5,
        "Java": 45.4,
        "Bash/Shell": 40.4,
        "Python": 37.9,
        "C#": 35.3,
        "PHP": 31.4,
        "C++": 24.6,
        "C": 22.1,
        "TypeScript": 18.3,
        "Ruby": 10.3,
        "Swift": 8.3,
        "Objective-C": 7.3,
        "Go": 7.2,
    }
}));
```

15.8 Graphics in a <canvas>

The `<canvas>` element has no appearance of its own but creates a drawing surface within the document and exposes a powerful drawing API to client-side JavaScript. The main difference between the `<canvas>` API and SVG is that with the canvas you create drawings by calling methods, and with SVG you create drawings by building a tree of XML elements. These two approaches are equivalently powerful: either one can be simulated with the other. On the surface, they are quite different, however, and each has its strengths and weaknesses. An SVG drawing, for example, is easily edited by removing elements from its description. To remove an element from the same graphic in a `<canvas>`, it is often necessary to erase the drawing and redraw it from scratch. Since the Canvas drawing API is JavaScript-based and relatively compact (unlike the SVG grammar), it is documented in more detail in this book.

3D Graphics in a Canvas

You can also call `getContext()` with the string "webgl" to obtain a context object that allows you to draw 3D graphics using the WebGL API. WebGL is a large, complicated, and low-level API that allows JavaScript programmers to access the GPU, write custom shaders, and perform other very powerful graphics operations. WebGL is not documented in this book, however: web developers are more likely to use utility libraries built on top of WebGL than to use the WebGL API directly.

Most of the Canvas drawing API is defined not on the `<canvas>` element itself, but instead on a "drawing context" object obtained with the `getContext()` method of the canvas. Call `getContext()` with the argument "2d" to obtain a CanvasRenderingContext2D object that you can use to draw two-dimensional graphics into the canvas.

As a simple example of the Canvas API, the following HTML document uses `<canvas>` elements and some JavaScript to display two simple shapes:

```
<p>This is a red square: <canvas id="square" width=10 height=10></canvas>.
<p>This is a blue circle: <canvas id="circle" width=10 height=10></canvas>.
<script>
let canvas = document.querySelector("#square");  // Get first canvas element
let context = canvas.getContext("2d");           // Get 2D drawing context
context.fillStyle = "#f00";                      // Set fill color to red
context.fillRect(0,0,10,10);                     // Fill a square

canvas = document.querySelector("#circle");      // Second canvas element
context = canvas.getContext("2d");               // Get its context
context.beginPath();                             // Begin a new "path"
context.arc(5, 5, 5, 0, 2*Math.PI, true);        // Add a circle to the path
context.fillStyle = "#00f";                      // Set blue fill color
context.fill();                                  // Fill the path
</script>
```

We've seen that SVG describes complex shapes as a "path" of lines and curves that can be drawn or filled. The Canvas API also uses the notion of a path. Instead of describing a path as a string of letters and numbers, a path is defined by a series of method calls, such as the `beginPath()` and `arc()` invocations in the preceding code. Once a path is defined, other methods, such as `fill()`, operate on that path. Various properties of the context object, such as `fillStyle`, specify how these operations are performed.

The subsections that follow demonstrate the methods and properties of the 2D Canvas API. Much of the example code that follows operates on a variable `c`. This variable holds the CanvasRenderingContext2D object of the canvas, but the code to initialize that variable is sometimes not shown. In order to make these examples run, you would need to add HTML markup to define a canvas with appropriate `width` and `height` attributes, and then add code like this to initialize the variable `c`:

```
let canvas = document.querySelector("#my_canvas_id");
let c = canvas.getContext('2d');
```

15.8.1 Paths and Polygons

To draw lines on a canvas and to fill the areas enclosed by those lines, you begin by defining a *path*. A path is a sequence of one or more subpaths. A subpath is a sequence of two or more points connected by line segments (or, as we'll see later, by curve segments). Begin a new path with the `beginPath()` method. Begin a new

subpath with the `moveTo()` method. Once you have established the starting point of a subpath with `moveTo()`, you can connect that point to a new point with a straight line by calling `lineTo()`. The following code defines a path that includes two line segments:

```
c.beginPath();        // Start a new path
c.moveTo(100, 100);   // Begin a subpath at (100,100)
c.lineTo(200, 200);   // Add a line from (100,100) to (200,200)
c.lineTo(100, 200);   // Add a line from (200,200) to (100,200)
```

This code simply defines a path; it does not draw anything on the canvas. To draw (or "stroke") the two line segments in the path, call the `stroke()` method, and to fill the area defined by those line segments, call `fill()`:

```
c.fill();             // Fill a triangular area
c.stroke();           // Stroke two sides of the triangle
```

This code (along with some additional code to set line widths and fill colors) produced the drawing shown in Figure 15-7.

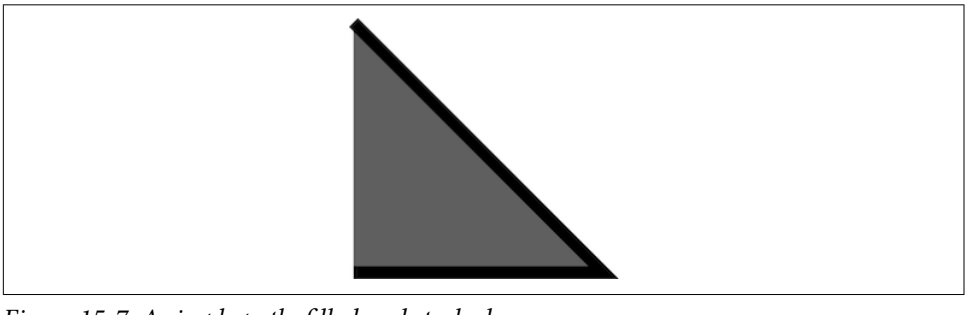

Figure 15-7. A simple path, filled and stroked

Notice that the subpath defined in Figure 15-7 is "open." It consists of just two line segments, and the end point is not connected back to the starting point. This means that it does not enclose a region. The `fill()` method fills open subpaths by acting as if a straight line connected the last point in the subpath to the first point in the subpath. That is why this code fills a triangle, but strokes only two sides of the triangle.

If you wanted to stroke all three sides of the triangle just shown, you would call the `closePath()` method to connect the end point of the subpath to the start point. (You could also call `lineTo(100,100)`, but then you end up with three line segments that share a start and end point but are not truly closed. When drawing with wide lines, the visual results are better if you use `closePath()`.)

There are two other important points to notice about `stroke()` and `fill()`. First, both methods operate on all subpaths in the current path. Suppose we had added another subpath in the preceding code:

```
    c.moveTo(300,100);    // Begin a new subpath at (300,100);
    c.lineTo(300,200);    // Draw a vertical line down to (300,200);
```

If we then called `stroke()`, we would draw two connected edges of a triangle and a disconnected vertical line.

The second point to note about `stroke()` and `fill()` is that neither one alters the current path: you can call `fill()` and the path will still be there when you call `stroke()`. When you are done with a path and want to begin another, you must remember to call `beginPath()`. If you don't, you'll end up adding new subpaths to the existing path, and you may end up drawing those old subpaths over and over again.

Example 15-5 defines a function for drawing regular polygons and demonstrates the use of `moveTo()`, `lineTo()`, and `closePath()` for defining subpaths and of `fill()` and `stroke()` for drawing those paths. It produces the drawing shown in Figure 15-8.

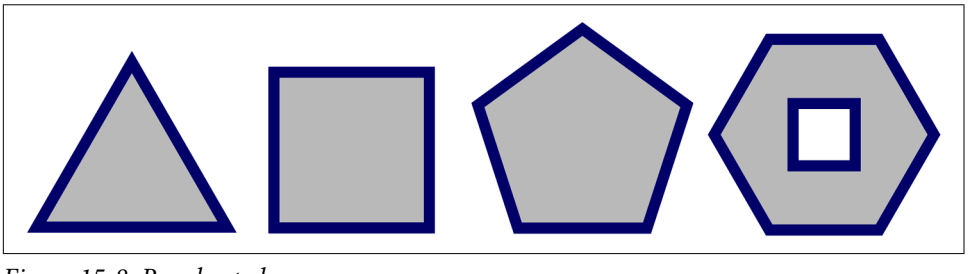

Figure 15-8. Regular polygons

Example 15-5. Regular polygons with moveTo(), lineTo(), and closePath()

```
// Define a regular polygon with n sides, centered at (x,y) with radius r.
// The vertices are equally spaced along the circumference of a circle.
// Put the first vertex straight up or at the specified angle.
// Rotate clockwise, unless the last argument is true.
function polygon(c, n, x, y, r, angle=0, counterclockwise=false) {
    c.moveTo(x + r*Math.sin(angle),  // Begin a new subpath at the first vertex
             y - r*Math.cos(angle)); // Use trigonometry to compute position
    let delta = 2*Math.PI/n;         // Angular distance between vertices
    for(let i = 1; i < n; i++) {     // For each of the remaining vertices
        angle += counterclockwise?-delta:delta; // Adjust angle
        c.lineTo(x + r*Math.sin(angle),         // Add line to next vertex
                 y - r*Math.cos(angle));
    }
    c.closePath();                   // Connect last vertex back to the first
}

// Assume there is just one canvas, and get its context object to draw with.
let c = document.querySelector("canvas").getContext("2d");

// Start a new path and add polygon subpaths
c.beginPath();
```

```
polygon(c, 3, 50, 70, 50);                  // Triangle
polygon(c, 4, 150, 60, 50, Math.PI/4);      // Square
polygon(c, 5, 255, 55, 50);                 // Pentagon
polygon(c, 6, 365, 53, 50, Math.PI/6);      // Hexagon
polygon(c, 4, 365, 53, 20, Math.PI/4, true); // Small square inside the hexagon

// Set some properties that control how the graphics will look
c.fillStyle = "#ccc";    // Light gray interiors
c.strokeStyle = "#008";  // outlined with dark blue lines
c.lineWidth = 5;         // five pixels wide.

// Now draw all the polygons (each in its own subpath) with these calls
c.fill();                // Fill the shapes
c.stroke();              // And stroke their outlines
```

Notice that this example draws a hexagon with a square inside it. The square and the hexagon are separate subpaths, but they overlap. When this happens (or when a single subpath intersects itself), the canvas needs to be able to determine which regions are inside the path and which are outside. The canvas uses a test known as the "nonzero winding rule" to achieve this. In this case, the interior of the square is not filled because the square and the hexagon were drawn in the opposite directions: the vertices of the hexagon were connected with line segments moving clockwise around the circle. The vertices of the square were connected counterclockwise. Had the square been drawn clockwise as well, the call to `fill()` would have filled the interior of the square as well.

15.8.2 Canvas Dimensions and Coordinates

The `width` and `height` attributes of the `<canvas>` element and the corresponding `width` and `height` properties of the Canvas object specify the dimensions of the canvas. The default canvas coordinate system places the origin (0,0) at the upper-left corner of the canvas. The *x* coordinates increase to the right and the *y* coordinates increase as you go down the screen. Points on the canvas can be specified using floating-point values.

The dimensions of a canvas cannot be altered without completely resetting the canvas. Setting either the `width` or `height` properties of a Canvas (even setting them to their current value) clears the canvas, erases the current path, and resets all graphics attributes (including current transformation and clipping region) to their original state.

The `width` and `height` attributes of a canvas specify the actual number of pixels that the canvas can draw into. Four bytes of memory are allocated for each pixel, so if `width` and `height` are both set to 100, the canvas allocates 40,000 bytes to represent 10,000 pixels.

The `width` and `height` attributes also specify the default size (in CSS pixels) at which the canvas will be displayed on the screen. If `window.devicePixelRatio` is 2, then 100 × 100 CSS pixels is actually 40,000 hardware pixels. When the contents of the canvas are drawn onto the screen, the 10,000 pixels in memory will need to be enlarged to cover 40,000 physical pixels on the screen, and this means that your graphics will not be as crisp as they could be.

For optimum image quality, you should not use the `width` and `height` attributes to set the on-screen size of the canvas. Instead, set the desired on-screen size CSS pixel size of the canvas with CSS `width` and `height` style attributes. Then, before you begin drawing in your JavaScript code, set the `width` and `height` properties of the canvas object to the number of CSS pixels times `window.devicePixelRatio`. Continuing with the preceding example, this technique would result in the canvas being displayed at 100 × 100 CSS pixels but allocating memory for 200 × 200 pixels. (Even with this technique, the user can zoom in on the canvas and may see fuzzy or pixelated graphics if they do. This is in contrast to SVG graphics, which remain crisp no matter the on-screen size or zoom level.)

15.8.3 Graphics Attributes

Example 15-5 set the properties `fillStyle`, `strokeStyle`, and `lineWidth` on the context object of the canvas. These properties are graphics attributes that specify the color to be used by `fill()` and by `stroke()`, and the width of the lines to be drawn by `stroke()`. Notice that these parameters are not passed to the `fill()` and `stroke()` methods, but are instead part of the general *graphics state* of the canvas. If you define a method that draws a shape and do not set these properties yourself, the caller of your method can define the color of the shape by setting the `strokeStyle` and `fillStyle` properties before calling your method. This separation of graphics state from drawing commands is fundamental to the Canvas API and is akin to the separation of presentation from content achieved by applying CSS stylesheets to HTML documents.

There are a number of properties (and also some methods) on the context object that affect the graphics state of the canvas. They are detailed below.

Line styles

The `lineWidth` property specifies how wide (in CSS pixels) the lines drawn by `stroke()` will be. The default value is 1. It is important to understand that line width is determined by the `lineWidth` property at the time `stroke()` is called, not at the time that `lineTo()` and other path-building methods are called. To fully understand the `lineWidth` property, it is important to visualize paths as infinitely thin one-dimensional lines. The lines and curves drawn by the `stroke()` method are centered over the path, with half of the `lineWidth` on either side. If you're stroking a closed

path and only want the line to appear outside the path, stroke the path first, then fill with an opaque color to hide the portion of the stroke that appears inside the path. Or if you only want the line to appear inside a closed path, call the `save()` and `clip()` methods first, then call `stroke()` and `restore()`. (The `save()`, `restore()`, and `clip()` methods are described later.)

When drawing lines that are more than about two pixels wide, the `lineCap` and `lineJoin` properties can have a significant impact on the visual appearance of the ends of a path and the vertices at which two path segments meet. Figure 15-9 illustrates the values and resulting graphical appearance of `lineCap` and `lineJoin`.

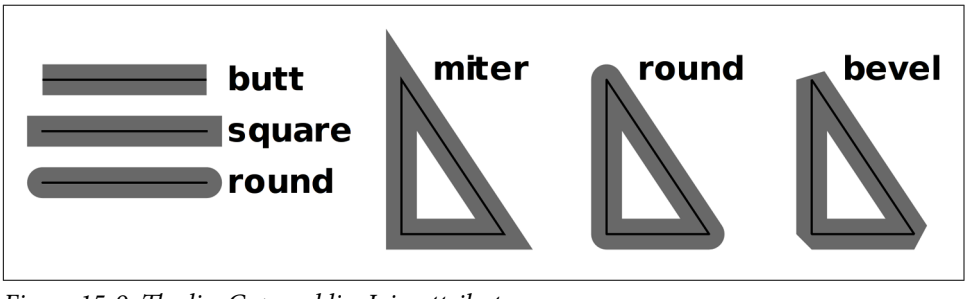

Figure 15-9. The lineCap and lineJoin attributes

The default value for `lineCap` is "butt." The default value for `lineJoin` is "miter." Note, however, that if two lines meet at a very narrow angle, then the resulting miter can become quite long and visually distracting. If the miter at a given vertex would be longer than half of the line width times the `miterLimit` property, that vertex will be drawn with a beveled join instead of a mitered join. The default value for `miterLimit` is 10.

The `stroke()` method can draw dashed and dotted lines as well as solid lines, and a canvas's graphics state includes an array of numbers that serves as a "dash pattern" by specifying how many pixels to draw, then how many to omit. Unlike other line-drawing properties, the dash pattern is set and queried with the methods `setLineDash()` and `getLineDash()` instead of with a property. To specify a dotted dash pattern, you might use `setLineDash()` like this:

```
c.setLineDash([18, 3, 3, 3]); // 18px dash, 3px space, 3px dot, 3px space
```

Finally, the `lineDashOffset` property specifies how far into the dash pattern drawing should begin. The default is 0. Paths stroked with the dash pattern shown here begin with an 18-pixel dash, but if `lineDashOffset` is set to 21, then that same path would begin with a dot followed by a space and a dash.

Colors, patterns, and gradients

The `fillStyle` and `strokeStyle` properties specify how paths are filled and stroked. The word "style" often means color, but these properties can also be used to specify a color gradient or an image to be used for filling and stroking. (Note that drawing a line is basically the same as filling a narrow region on both sides of the line, and filling and stroking are fundamentally the same operation.)

If you want to fill or stroke with a solid color (or a translucent color), simply set these properties to a valid CSS color string. Nothing else is required.

To fill (or stroke) with a color gradient, set `fillStyle` (or `strokeStyle`) to a CanvasGradient object returned by the `createLinearGradient()` or `createRadialGradient()` methods of the context. The arguments to `createLinearGradient()` are the coordinates of two points that define a line (it does not need to be horizontal or vertical) along which the colors will vary. The arguments to `createRadialGradient()` specify the centers and radii of two circles. (They need not be concentric, but the first circle typically lies entirely inside the second.) Areas inside the smaller circle or outside the larger will be filled with solid colors; areas between the two will be filled with a color gradient.

After creating the CanvasGradient object that defines the regions of the canvas that will be filled, you must define the gradient colors by calling the `addColorStop()` method of the CanvasGradient. The first argument to this method is a number between 0.0 and 1.0. The second argument is a CSS color specification. You must call this method at least twice to define a simple color gradient, but you may call it more than that. The color at 0.0 will appear at the start of the gradient, and the color at 1.0 will appear at the end. If you specify additional colors, they will appear at the specified fractional position within the gradient. Between the points you specify, colors will be smoothly interpolated. Here are some examples:

```
// A linear gradient, diagonally across the canvas (assuming no transforms)
let bgfade = c.createLinearGradient(0,0,canvas.width,canvas.height);
bgfade.addColorStop(0.0, "#88f");  // Start with light blue in upper left
bgfade.addColorStop(1.0, "#fff");  // Fade to white in lower right

// A gradient between two concentric circles. Transparent in the middle
// fading to translucent gray and then back to transparent.
let donut = c.createRadialGradient(300,300,100, 300,300,300);
donut.addColorStop(0.0, "transparent");           // Transparent
donut.addColorStop(0.7, "rgba(100,100,100,.9)");  // Translucent gray
donut.addColorStop(1.0, "rgba(0,0,0,0)");         // Transparent again
```

An important point to understand about gradients is that they are not position-independent. When you create a gradient, you specify bounds for the gradient. If you then attempt to fill an area outside of those bounds, you'll get the solid color defined at one end or the other of the gradient.

In addition to colors and color gradients, you can also fill and stroke using images. To do this, set `fillStyle` or `strokeStyle` to a CanvasPattern returned by the `createPattern()` method of the context object. The first argument to this method should be an `<img>` or `<canvas>` element that contains the image you want to fill or stroke with. (Note that the source image or canvas does not need to be inserted into the document in order to be used in this way.) The second argument to `createPattern()` is the string "repeat," "repeat-x," "repeat-y," or "no-repeat," which specifies whether (and in which dimensions) the background images repeat.

Text styles

The `font` property specifies the font to be used by the text-drawing methods `fillText()` and `strokeText()` (see "Text" on page 498). The value of the `font` property should be a string in the same syntax as the CSS `font` attribute.

The `textAlign` property specifies how the text should be horizontally aligned with respect to the X coordinate passed to `fillText()` or `strokeText()`. Legal values are "start," "left," "center," "right," and "end." The default is "start," which, for left-to-right text, has the same meaning as "left."

The `textBaseline` property specifies how the text should be vertically aligned with respect to the *y* coordinate. The default value is "alphabetic," and it is appropriate for Latin and similar scripts. The value "ideographic" is intended for use with scripts such as Chinese and Japanese. The value "hanging" is intended for use with Devanagari and similar scripts (which are used for many of the languages of India). The "top," "middle," and "bottom" baselines are purely geometric baselines, based on the "em square" of the font.

Shadows

Four properties of the context object control the drawing of drop shadows. If you set these properties appropriately, any line, area, text, or image you draw will be given a shadow, which will make it appear as if it is floating above the canvas surface.

The `shadowColor` property specifies the color of the shadow. The default is fully transparent black, and shadows will never appear unless you set this property to a translucent or opaque color. This property can only be set to a color string: patterns and gradients are not allowed for shadows. Using a translucent shadow color produces the most realistic shadow effects because it allows the background to show through.

The `shadowOffsetX` and `shadowOffsetY` properties specify the X and Y offsets of the shadow. The default for both properties is 0, which places the shadow directly beneath your drawing, where it is not visible. If you set both properties to a positive value, shadows will appear below and to the right of what you draw, as if there were a

light source above and to the left, shining onto the canvas from outside the computer screen. Larger offsets produce larger shadows and make drawn objects appear as if they are floating "higher" above the canvas. These values are not affected by coordinate transformations (§15.8.5): shadow direction and "height" remain consistent even when shapes are rotated and scaled.

The `shadowBlur` property specifies how blurred the edges of the shadow are. The default value is 0, which produces crisp, unblurred shadows. Larger values produce more blur, up to an implementation-defined upper bound.

Translucency and compositing

If you want to stroke or fill a path using a translucent color, you can set `strokeStyle` or `fillStyle` using a CSS color syntax like "rgba(…)" that supports alpha transparency. The "a" in "RGBA" stands for "alpha" and is a value between 0 (fully transparent) and 1 (fully opaque). But the Canvas API provides another way to work with translucent colors. If you do not want to explicitly specify an alpha channel for each color, or if you want to add translucency to opaque images or patterns, you can set the `globalAlpha` property. Every pixel you draw will have its alpha value multiplied by `globalAlpha`. The default is 1, which adds no transparency. If you set `globalAlpha` to 0, everything you draw will be fully transparent, and nothing will appear in the canvas. But if you set this property to 0.5, then pixels that would otherwise have been opaque will be 50% opaque, and pixels that would have been 50% opaque will be 25% opaque instead.

When you stroke lines, fill regions, draw text, or copy images, you generally expect the new pixels to be drawn on top of the pixels that are already in the canvas. If you are drawing opaque pixels, they simply replace the pixels that are already there. If you are drawing with translucent pixels, the new ("source") pixel is combined with the old ("destination") pixel so that the old pixel shows through the new pixel based on how transparent that pixel is.

This process of combining new (possibly translucent) source pixels with existing (possibly translucent) destination pixels is called *compositing*, and the compositing process described previously is the default way that the Canvas API combines pixels. But you can set the `globalCompositeOperation` property to specify other ways of combining pixels. The default value is "source-over," which means that source pixels are drawn "over" the destination pixels and are combined with them if the source is translucent. But if you set `globalCompositeOperation` to "destination-over", then the canvas will combine pixels as if the new source pixels were drawn beneath the existing destination pixels. If the destination is translucent or transparent, some or all of the source pixel color is visible in the resulting color. As another example, the compositing mode "source-atop" combines the source pixels with the transparency of the destination pixels so that nothing is drawn on portions of the canvas that are already

fully transparent. There are a number of legal values for `globalCompositeOperation`, but most have only specialized uses and are not covered here.

Saving and restoring graphics state

Since the Canvas API defines graphics attributes on the context object, you might be tempted to call `getContext()` multiple times to obtain multiple context objects. If you could do this, you could define different attributes on each context: each context would then be like a different brush and would paint with a different color or draw lines of different widths. Unfortunately, you cannot use the canvas in this way. Each `<canvas>` element has only a single context object, and every call to `getContext()` returns the same CanvasRenderingContext2D object.

Although the Canvas API only allows you to define a single set of graphics attributes at a time, it does allow you to save the current graphics state so that you can alter it and then easily restore it later. The `save()` method pushes the current graphics state onto a stack of saved states. The `restore()` method pops the stack and restores the most recently saved state. All of the properties that have been described in this section are part of the saved state, as are the current transformation and clipping region (both of which are explained later). Importantly, the currently defined path and the current point are not part of the graphics state and cannot be saved and restored.

15.8.4 Canvas Drawing Operations

We've already seen some basic canvas methods—`beginPath()`, `moveTo()`, `lineTo()`, `closePath()`, `fill()`, and `stroke()`—for defining, filling, and drawing lines and polygons. But the Canvas API includes other drawing methods as well.

Rectangles

CanvasRenderingContext2D defines four methods for drawing rectangles. All four of these rectangle methods expect two arguments that specify one corner of the rectangle followed by the rectangle width and height. Normally, you specify the upper-left corner and then pass a positive width and positive height, but you may also specify other corners and pass negative dimensions.

`fillRect()` fills the specified rectangle with the current `fillStyle`. `strokeRect()` strokes the outline of the specified rectangle using the current `strokeStyle` and other line attributes. `clearRect()` is like `fillRect()`, but it ignores the current fill style and fills the rectangle with transparent black pixels (the default color of all blank canvases). The important thing about these three methods is that they do not affect the current path or the current point within that path.

The final rectangle method is named `rect()`, and it does affect the current path: it adds the specified rectangle, in a subpath of its own, to the path. Like other path-definition methods, it does not fill or stroke anything itself.

Curves

A path is a sequence of subpaths, and a subpath is a sequence of connected points. In the paths we defined in §15.8.1, those points were connected with straight line segments, but that need not always be the case. The CanvasRenderingContext2D object defines a number of methods that add a new point to the subpath and connect the current point to that new point with a curve:

`arc()`
: This method adds a circle, or a portion of a circle (an arc), to the path. The arc to be drawn is specified with six parameters: the *x* and *y* coordinates of the center of a circle, the radius of the circle, the start and end angles of the arc, and the direction (clockwise or counterclockwise) of the arc between those two angles. If there is a current point in the path, then this method connects the current point to the beginning of the arc with a straight line (which is useful when drawing wedges or pie slices), then connects the beginning of the arc to the end of the arc with a portion of a circle, leaving the end of the arc as the new current point. If there is no current point when this method is called, then it only adds the circular arc to the path.

`ellipse()`
: This method is much like `arc()` except that it adds an ellipse or a portion of an ellipse to the path. Instead of one radius, it has two: an *x*-axis radius and a *y*-axis radius. Also, because ellipses are not radially symmetrical, this method takes another argument that specifies the number of radians by which the ellipse is rotated clockwise about its center.

`arcTo()`
: This method draws a straight line and a circular arc just like the `arc()` method does, but it specifies the arc to be drawn using different parameters. The arguments to `arcTo()` specify points P1 and P2 and a radius. The arc that is added to the path has the specified radius. It begins at the tangent point with the (imaginary) line from the current point to P1 and ends at the tangent point with the (imaginary) line between P1 and P2. This unusual-seeming method of specifying arcs is actually quite useful for drawing shapes with rounded corners. If you specify a radius of 0, this method just draws a straight line from the current point to P1. With a nonzero radius, however, it draws a straight line from the current point in the direction of P1, then curves that line around in a circle until it is heading in the direction of P2.

`bezierCurveTo()`
: This method adds a new point P to the subpath and connects it to the current point with a cubic Bezier curve. The shape of the curve is specified by two "control points," C1 and C2. At the start of the curve (at the current point), the curve heads in the direction of C1. At the end of the curve (at point P), the curve arrives from the direction of C2. In between these points, the direction of the curve varies smoothly. The point P becomes the new current point for the subpath.

`quadraticCurveTo()`
: This method is like `bezierCurveTo()`, but it uses a quadratic Bezier curve instead of a cubic Bezier curve and has only a single control point.

You can use these methods to draw paths like those in Figure 15-10.

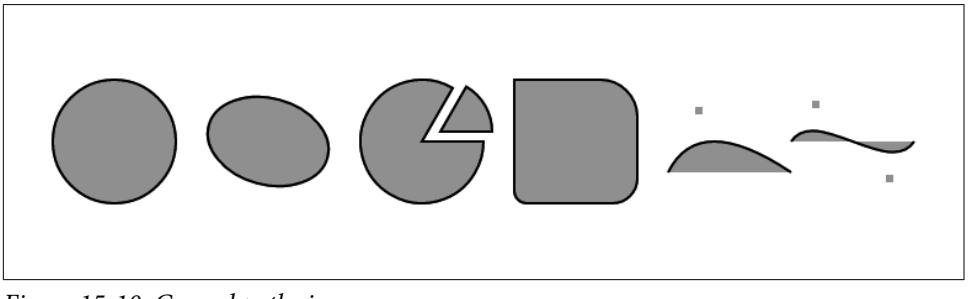

Figure 15-10. Curved paths in a canvas

Example 15-6 shows the code used to create Figure 15-10. The methods demonstrated in this code are some of the most complicated in the Canvas API; consult an online reference for complete details on the methods and their arguments.

Example 15-6. Adding curves to a path

```
// A utility function to convert angles from degrees to radians
function rads(x) { return Math.PI*x/180; }

// Get the context object of the document's canvas element
let c = document.querySelector("canvas").getContext("2d");

// Define some graphics attributes and draw the curves
c.fillStyle = "#aaa";     // Gray fills
c.lineWidth = 2;          // 2-pixel black (by default) lines

// Draw a circle.
// There is no current point, so draw just the circle with no straight
// line from the current point to the start of the circle.
c.beginPath();
c.arc(75,100,50,          // Center at (75,100), radius 50
```

```
      0,rads(360),false); // Go clockwise from 0 to 360 degrees
c.fill();                 // Fill the circle
c.stroke();               // Stroke its outline.

// Now draw an ellipse in the same way
c.beginPath();            // Start new path not connected to the circle
c.ellipse(200, 100, 50, 35, rads(15),  // Center, radii, and rotation
          0, rads(360), false);        // Start angle, end angle, direction

// Draw a wedge. Angles are measured clockwise from the positive x axis.
// Note that arc() adds a line from the current point to the arc start.
c.moveTo(325, 100);       // Start at the center of the circle.
c.arc(325, 100, 50,       // Circle center and radius
      rads(-60), rads(0), // Start at angle -60 and go to angle 0
      true);              // counterclockwise
c.closePath();            // Add radius back to the center of the circle

// Similar wedge, offset a bit, and in the opposite direction
c.moveTo(340, 92);
c.arc(340, 92, 42, rads(-60), rads(0), false);
c.closePath();

// Use arcTo() for rounded corners. Here we draw a square with
// upper left corner at (400,50) and corners of varying radii.
c.moveTo(450, 50);          // Begin in the middle of the top edge.
c.arcTo(500,50,500,150,30); // Add part of top edge and upper right corner.
c.arcTo(500,150,400,150,20); // Add right edge and lower right corner.
c.arcTo(400,150,400,50,10); // Add bottom edge and lower left corner.
c.arcTo(400,50,500,50,0);   // Add left edge and upper left corner.
c.closePath();              // Close path to add the rest of the top edge.

// Quadratic Bezier curve: one control point
c.moveTo(525, 125);                     // Begin here
c.quadraticCurveTo(550, 75, 625, 125);  // Draw a curve to (625, 125)
c.fillRect(550-3, 75-3, 6, 6);          // Mark the control point (550,75)

// Cubic Bezier curve
c.moveTo(625, 100);                     // Start at (625, 100)
c.bezierCurveTo(645,70,705,130,725,100); // Curve to (725, 100)
c.fillRect(645-3, 70-3, 6, 6);          // Mark control points
c.fillRect(705-3, 130-3, 6, 6);

// Finally, fill the curves and stroke their outlines.
c.fill();
c.stroke();
```

Text

To draw text in a canvas, you normally use the `fillText()` method, which draws text using the color (or gradient or pattern) specified by the `fillStyle` property. For special effects at large text sizes, you can use `strokeText()` to draw the outline of the individual font glyphs. Both methods take the text to be drawn as their first argument and take the *x* and *y* coordinates of the text as the second and third arguments. Neither method affects the current path or the current point.

`fillText()` and `strokeText()` take an optional fourth argument. If given, this argument specifies the maximum width of the text to be displayed. If the text would be wider than the specified value when drawn using the `font` property, the canvas will make it fit by scaling it or by using a narrower or smaller font.

If you need to measure text yourself before drawing it, pass it to the `measureText()` method. This method returns a TextMetrics object that specifies the measurements of the text when drawn with the current `font`. At the time of this writing, the only "metric" contained in the `TextMetrics` object is the width. Query the on-screen width of a string like this:

```
let width = c.measureText(text).width;
```

This is useful if you want to center a string of text within a canvas, for example.

Images

In addition to vector graphics (paths, lines, etc.), the Canvas API also supports bitmap images. The `drawImage()` method copies the pixels of a source image (or of a rectangle within the source image) onto the canvas, scaling and rotating the pixels of the image as necessary.

`drawImage()` can be invoked with three, five, or nine arguments. In all cases, the first argument is the source image from which pixels are to be copied. This image argument is often an `<img>` element, but it can also be another `<canvas>` element or even a `<video>` element (from which a single frame will be copied). If you specify an `<img>` or `<video>` element that is still loading its data, the `drawImage()` call will do nothing.

In the three-argument version of `drawImage()`, the second and third arguments specify the *x* and *y* coordinates at which the upper-left corner of the image is to be drawn. In this version of the method, the entire source image is copied to the canvas. The *x* and *y* coordinates are interpreted in the current coordinate system, and the image is scaled and rotated if necessary, depending on the canvas transform currently in effect.

The five-argument version of `drawImage()` adds `width` and `height` arguments to the `x` and `y` arguments described earlier. These four arguments define a destination rectangle within the canvas. The upper-left corner of the source image goes at `(x,y)`, and the lower-right corner goes at `(x+width, y+height)`. Again, the entire source

image is copied. With this version of the method, the source image will be scaled to fit the destination rectangle.

The nine-argument version of `drawImage()` specifies both a source rectangle and a destination rectangle and copies only the pixels within the source rectangle. Arguments two through five specify the source rectangle. They are measured in CSS pixels. If the source image is another canvas, the source rectangle uses the default coordinate system for that canvas and ignores any transformations that have been specified. Arguments six through nine specify the destination rectangle into which the image is drawn and are in the current coordinate system of the canvas, not in the default coordinate system.

In addition to drawing images into a canvas, we can also extract the content of a canvas as an image using the `toDataURL()` method. Unlike all the other methods described here, `toDataURL()` is a method of the Canvas element itself, not of the context object. You normally invoke `toDataURL()` with no arguments, and it returns the content of the canvas as a PNG image, encoded as a string using a `data:` URL. The returned URL is suitable for use with an `<img>` element, and you can make a static snapshot of a canvas with code like this:

```
let img = document.createElement("img");  // Create an <img> element
img.src = canvas.toDataURL();             // Set its src attribute
document.body.appendChild(img);           // Append it to the document
```

15.8.5 Coordinate System Transforms

As we've noted, the default coordinate system of a canvas places the origin in the upper-left corner, has *x* coordinates increasing to the right, and has *y* coordinates increasing downward. In this default system, the coordinates of a point map directly to a CSS pixel (which then maps directly to one or more device pixels). Certain canvas operations and attributes (such as extracting raw pixel values and setting shadow offsets) always use this default coordinate system. In addition to the default coordinate system, however, every canvas has a "current transformation matrix" as part of its graphics state. This matrix defines the current coordinate system of the canvas. In most canvas operations, when you specify the coordinates of a point, it is taken to be a point in the current coordinate system, not in the default coordinate system. The current transformation matrix is used to convert the coordinates you specified to the equivalent coordinates in the default coordinate system.

The `setTransform()` method allows you to set a canvas's transformation matrix directly, but coordinate system transformations are usually easier to specify as a sequence of translations, rotations, and scaling operations. Figure 15-11 illustrates these operations and their effect on the canvas coordinate system. The program that produced the figure drew the same set of axes seven times in a row. The only thing

that changed each time was the current transform. Notice that the transforms affect the text as well as the lines that are drawn.

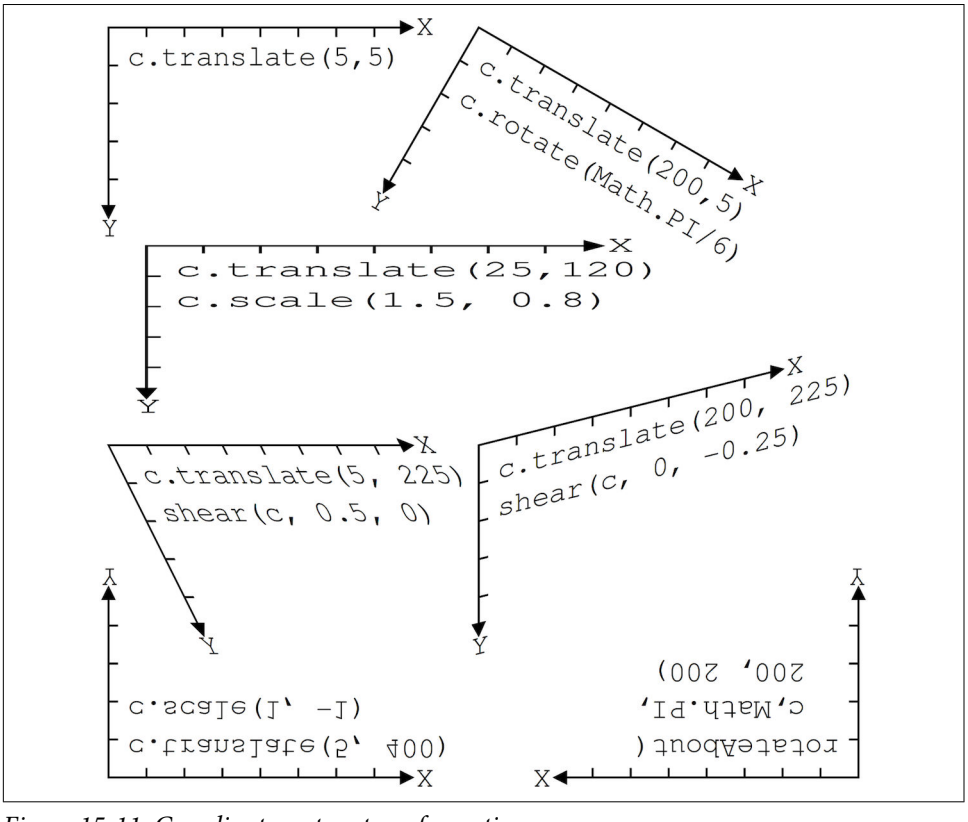

Figure 15-11. Coordinate system transformations

The `translate()` method simply moves the origin of the coordinate system left, right, up, or down. The `rotate()` method rotates the axes clockwise by the specified angle. (The Canvas API always specifies angles in radians. To convert degrees to radians, divide by 180 and multiply by `Math.PI`.) The `scale()` method stretches or contracts distances along the *x* or *y* axes.

Passing a negative scale factor to the `scale()` method flips that axis across the origin, as if it were reflected in a mirror. This is what was done in the lower left of Figure 15-11: `translate()` was used to move the origin to the bottom-left corner of the canvas, then `scale()` was used to flip the *y* axis around so that *y* coordinates increase as we go up the page. A flipped coordinate system like this is familiar from algebra class and may be useful for plotting data points on charts. Note, however, that it makes text difficult to read!

Understanding transformations mathematically

I find it easiest to understand transforms geometrically, thinking about `translate()`, `rotate()`, and `scale()` as transforming the axes of the coordinate system as illustrated in Figure 15-11. It is also possible to understand transforms algebraically as equations that map the coordinates of a point `(x,y)` in the transformed coordinate system back to the coordinates `(x',y')` of the same point in the previous coordinate system.

The method call `c.translate(dx,dy)` can be described with these equations:

```
x' = x + dx;  // An X coordinate of 0 in the new system is dx in the old
y' = y + dy;
```

Scaling operations have similarly simple equations. A call `c.scale(sx,sy)` can be described like this:

```
x' = sx * x;
y' = sy * y;
```

Rotations are more complicated. The call `c.rotate(a)` is described by these trigonometric equations:

```
x' = x * cos(a) - y * sin(a);
y' = y * cos(a) + x * sin(a);
```

Notice that the order of transformations matters. Suppose we start with the default coordinate system of a canvas, then translate it, and then scale it. In order to map the point `(x,y)` in the current coordinate system back to the point `(x'',y'')` in the default coordinate system, we must first apply the scaling equations to map the point to an intermediate point `(x',y')` in the translated but unscaled coordinate system, then use the translation equations to map from this intermediate point to `(x'',y'')`. The result is this:

```
x'' = sx*x + dx;
y'' = sy*y + dy;
```

If, on the other hand, we'd called `scale()` before calling `translate()`, the resulting equations would be different:

```
x'' = sx*(x + dx);
y'' = sy*(y + dy);
```

The key thing to remember when thinking algebraically about sequences of transformations is that you must work backward from the last (most recent) transformation to the first. When thinking geometrically about transformed axes, however, you work forward from first transformation to last.

The transformations supported by the canvas are known as *affine transforms*. Affine transforms may modify the distances between points and the angles between lines, but parallel lines always remain parallel after an affine transformation—it is not possible, for example, to specify a fish-eye lens distortion with an affine transform. An

arbitrary affine transform can be described by the six parameters `a` through `f` in these equations:

```
x' = ax + cy + e
y' = bx + dy + f
```

You can apply an arbitrary transformation to the current coordinate system by passing those six parameters to the `transform()` method. Figure 15-11 illustrates two types of transformations—shears and rotations about a specified point—that you can implement with the `transform()` method like this:

```
// Shear transform:
//   x' = x + kx*y;
//   y' = ky*x + y;
function shear(c, kx, ky) { c.transform(1, ky, kx, 1, 0, 0); }

// Rotate theta radians counterclockwise around the point (x,y)
// This can also be accomplished with a translate, rotate, translate sequence
function rotateAbout(c, theta, x, y) {
    let ct = Math.cos(theta);
    let st = Math.sin(theta);
    c.transform(ct, -st, st, ct, -x*ct-y*st+x, x*st-y*ct+y);
}
```

The `setTransform()` method takes the same arguments as `transform()`, but instead of transforming the current coordinate system, it ignores the current system, transforms the default coordinate system, and makes the result the new current coordinate system. `setTransform()` is useful to temporarily reset the canvas to its default coordinate system:

```
c.save();                       // Save current coordinate system
c.setTransform(1,0,0,1,0,0);    // Revert to the default coordinate system
// Perform operations using default CSS pixel coordinates
c.restore();                    // Restore the saved coordinate system
```

Transformation example

Example 15-7 demonstrates the power of coordinate system transformations by using the `translate()`, `rotate()`, and `scale()` methods recursively to draw a Koch snowflake fractal. The output of this example appears in Figure 15-12, which shows Koch snowflakes with 0, 1, 2, 3, and 4 levels of recursion.

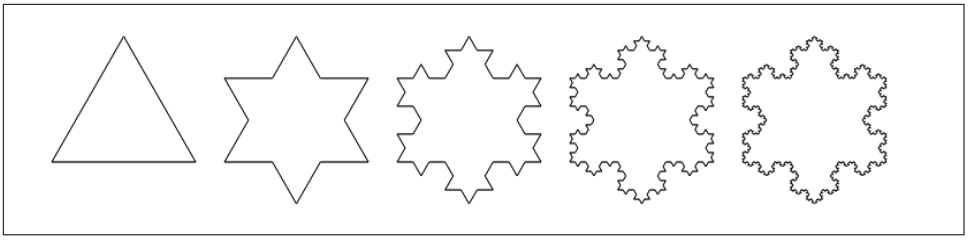

Figure 15-12. Koch snowflakes

The code that produces these figures is elegant, but its use of recursive coordinate system transformations makes it somewhat difficult to understand. Even if you don't follow all the nuances, note that the code includes only a single invocation of the `lineTo()` method. Every single line segment in Figure 15-12 is drawn like this:

```
c.lineTo(len, 0);
```

The value of the variable `len` does not change during the execution of the program, so the position, orientation, and length of each of the line segments is determined by translations, rotations, and scaling operations.

Example 15-7. A Koch snowflake with transformations

```
let deg = Math.PI/180;  // For converting degrees to radians

// Draw a level-n Koch snowflake fractal on the canvas context c,
// with lower-left corner at (x,y) and side length len.
function snowflake(c, n, x, y, len) {
    c.save();           // Save current transformation
    c.translate(x,y);   // Translate origin to starting point
    c.moveTo(0,0);      // Begin a new subpath at the new origin
    leg(n);             // Draw the first leg of the snowflake
    c.rotate(-120*deg); // Now rotate 120 degrees counterclockwise
    leg(n);             // Draw the second leg
    c.rotate(-120*deg); // Rotate again
    leg(n);             // Draw the final leg
    c.closePath();      // Close the subpath
    c.restore();        // And restore original transformation

    // Draw a single leg of a level-n Koch snowflake.
    // This function leaves the current point at the end of the leg it has
    // drawn and translates the coordinate system so the current point is (0,0).
    // This means you can easily call rotate() after drawing a leg.
    function leg(n) {
        c.save();               // Save the current transformation
        if (n === 0) {          // Nonrecursive case:
            c.lineTo(len, 0);   //   Just draw a horizontal line
        }                       //
        else {                  // Recursive case: draw 4 sub-legs like:  \/
```

```
            c.scale(1/3,1/3);   // Sub-legs are 1/3 the size of this leg
            leg(n-1);           // Recurse for the first sub-leg
            c.rotate(60*deg);   // Turn 60 degrees clockwise
            leg(n-1);           // Second sub-leg
            c.rotate(-120*deg); // Rotate 120 degrees back
            leg(n-1);           // Third sub-leg
            c.rotate(60*deg);   // Rotate back to our original heading
            leg(n-1);           // Final sub-leg
        }
        c.restore();            // Restore the transformation
        c.translate(len, 0);    // But translate to make end of leg (0,0)
    }
}

let c = document.querySelector("canvas").getContext("2d");
snowflake(c, 0, 25, 125, 125);  // A level-0 snowflake is a triangle
snowflake(c, 1, 175, 125, 125); // A level-1 snowflake is a 6-sided star
snowflake(c, 2, 325, 125, 125); // etc.
snowflake(c, 3, 475, 125, 125);
snowflake(c, 4, 625, 125, 125); // A level-4 snowflake looks like a snowflake!
c.stroke();                     // Stroke this very complicated path
```

15.8.6 Clipping

After defining a path, you usually call `stroke()` or `fill()` (or both). You can also call the `clip()` method to define a clipping region. Once a clipping region is defined, nothing will be drawn outside of it. Figure 15-13 shows a complex drawing produced using clipping regions. The vertical stripe running down the middle and the text along the bottom of the figure were stroked with no clipping region and then filled after the triangular clipping region was defined.

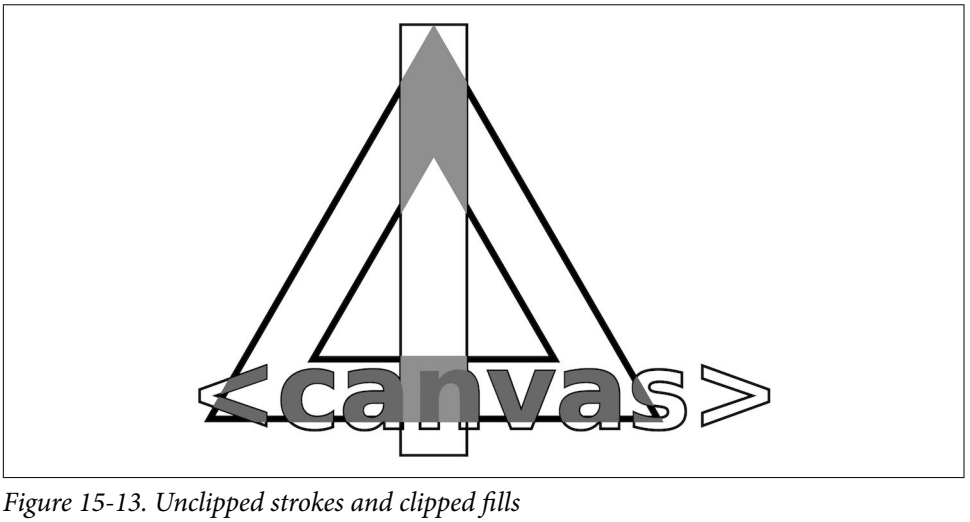

Figure 15-13. Unclipped strokes and clipped fills

Figure 15-13 was generated using the `polygon()` method of Example 15-5 and the following code:

```
// Define some drawing attributes
c.font = "bold 60pt sans-serif";    // Big font
c.lineWidth = 2;                    // Narrow lines
c.strokeStyle = "#000";             // Black lines

// Outline a rectangle and some text
c.strokeRect(175, 25, 50, 325);     // A vertical stripe down the middle
c.strokeText("<canvas>", 15, 330);  // Note strokeText() instead of fillText()

// Define a complex path with an interior that is outside.
polygon(c,3,200,225,200);           // Large triangle
polygon(c,3,200,225,100,0,true);    // Smaller reverse triangle inside

// Make that path the clipping region.
c.clip();

// Stroke the path with a 5 pixel line, entirely inside the clipping region.
c.lineWidth = 10;       // Half of this 10 pixel line will be clipped away
c.stroke();

// Fill the parts of the rectangle and text that are inside the clipping region
c.fillStyle = "#aaa";             // Light gray
c.fillRect(175, 25, 50, 325);     // Fill the vertical stripe
c.fillStyle = "#888";             // Darker gray
c.fillText("<canvas>", 15, 330);  // Fill the text
```

It is important to note that when you call `clip()`, the current path is itself clipped to the current clipping region, then that clipped path becomes the new clipping region. This means that the `clip()` method can shrink the clipping region but can never enlarge it. There is no method to reset the clipping region, so before calling `clip()`, you should typically call `save()` so that you can later `restore()` the unclipped region.

15.8.7 Pixel Manipulation

The `getImageData()` method returns an ImageData object that represents the raw pixels (as R, G, B, and A components) from a rectangular region of your canvas. You can create empty `ImageData` objects with `createImageData()`. The pixels in an ImageData object are writable, so you can set them any way you want, then copy those pixels back onto the canvas with `putImageData()`.

These pixel manipulation methods provide very low-level access to the canvas. The rectangle you pass to `getImageData()` is in the default coordinate system: its dimensions are measured in CSS pixels, and it is not affected by the current transformation. When you call `putImageData()`, the position you specify is also measured in the default coordinate system. Furthermore, `putImageData()` ignores all graphics

attributes. It does not perform any compositing, it does not multiply pixels by `globalAlpha`, and it does not draw shadows.

Pixel manipulation methods are useful for implementing image processing. Example 15-8 shows how to create a simple motion blur or "smear" effect like that shown in Figure 15-14.

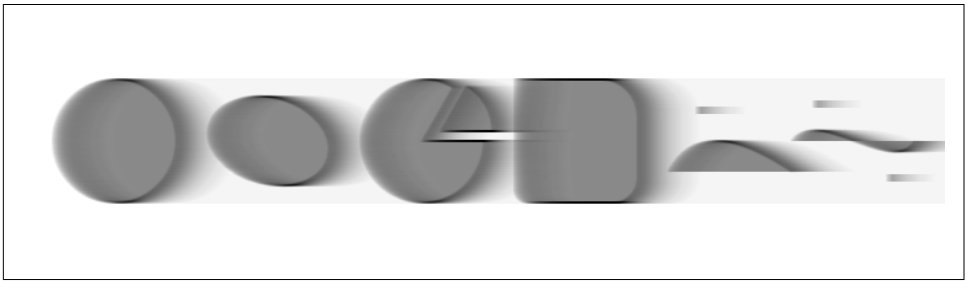

Figure 15-14. A motion blur effect created by image processing

The following code demonstrates `getImageData()` and `putImageData()` and shows how to iterate through and modify the pixel values in an ImageData object.

Example 15-8. Motion blur with ImageData

```
// Smear the pixels of the rectangle to the right, producing a
// sort of motion blur as if objects are moving from right to left.
// n must be 2 or larger. Larger values produce bigger smears.
// The rectangle is specified in the default coordinate system.
function smear(c, n, x, y, w, h) {
    // Get the ImageData object that represents the rectangle of pixels to smear
    let pixels = c.getImageData(x, y, w, h);

    // This smear is done in-place and requires only the source ImageData.
    // Some image processing algorithms require an additional ImageData to
    // store transformed pixel values. If we needed an output buffer, we could
    // create a new ImageData with the same dimensions like this:
    //   let output_pixels = c.createImageData(pixels);

    // Get the dimensions of the grid of pixels in the ImageData object
    let width = pixels.width, height = pixels.height;

    // This is the byte array that holds the raw pixel data, left-to-right and
    // top-to-bottom. Each pixel occupies 4 consecutive bytes in R,G,B,A order.
    let data = pixels.data;

    // Each pixel after the first in each row is smeared by replacing it with
    // 1/nth of its own value plus m/nths of the previous pixel's value
    let m = n-1;

    for(let row = 0; row < height; row++) {  // For each row
```

```
            let i = row*width*4 + 4;  // The offset of the second pixel of the row
            for(let col = 1; col < width; col++, i += 4) { // For each column
                data[i] =   (data[i] + data[i-4]*m)/n;     // Red pixel component
                data[i+1] = (data[i+1] + data[i-3]*m)/n;   // Green
                data[i+2] = (data[i+2] + data[i-2]*m)/n;   // Blue
                data[i+3] = (data[i+3] + data[i-1]*m)/n;   // Alpha component
            }
        }

        // Now copy the smeared image data back to the same position on the canvas
        c.putImageData(pixels, x, y);
    }
```

15.9 Audio APIs

The HTML `<audio>` and `<video>` tags allow you to easily include sound and videos in your web pages. These are complex elements with significant APIs and nontrivial user interfaces. You can control media playback with the `play()` and `pause()` methods. You can set the `volume` and `playbackRate` properties to control the audio volume and speed of playback. And you can skip to a particular time within the media by setting the `currentTime` property.

We will not cover `<audio>` and `<video>` tags in any further detail here, however. The following subsections demonstrate two ways to add scripted sound effects to your web pages.

15.9.1 The Audio() Constructor

You don't have to include an `<audio>` tag in your HTML document in order to include sound effects in your web pages. You can dynamically create `<audio>` elements with the normal DOM `document.createElement()` method, or, as a shortcut, you can simply use the `Audio()` constructor. You do not have to add the created element to your document in order to play it. You can simply call its `play()` method:

```
// Load the sound effect in advance so it is ready for use
let soundeffect = new Audio("soundeffect.mp3");

// Play the sound effect whenever the user clicks the mouse button
document.addEventListener("click", () => {
    soundeffect.cloneNode().play(); // Load and play the sound
});
```

Note the use of `cloneNode()` here. If the user clicks the mouse rapidly, we want to be able to have multiple overlapping copies of the sound effect playing at the same time. To do that, we need multiple Audio elements. Because the Audio elements are not added to the document, they will be garbage collected when they are done playing.

15.9.2 The WebAudio API

In addition to playback of recorded sounds with Audio elements, web browsers also allow the generation and playback of synthesized sounds with the WebAudio API. Using the WebAudio API is like hooking up an old-style electronic synthesizer with patch cords. With WebAudio, you create a set of AudioNode objects, which represents sources, transformations, or destinations of waveforms, and then connect these nodes together into a network to produce sounds. The API is not particularly complex, but a full explanation requires an understanding of electronic music and signal processing concepts that are beyond the scope of this book.

The following code below uses the WebAudio API to synthesize a short chord that fades out over about a second. This example demonstrates the basics of the WebAudio API. If this is interesting to you, you can find much more about this API online:

```
// Begin by creating an audioContext object. Safari still requires
// us to use webkitAudioContext instead of AudioContext.
let audioContext = new (this.AudioContext||this.webkitAudioContext)();

// Define the base sound as a combination of three pure sine waves
let notes = [ 293.7, 370.0, 440.0 ]; // D major chord: D, F# and A

// Create oscillator nodes for each of the notes we want to play
let oscillators = notes.map(note => {
    let o = audioContext.createOscillator();
    o.frequency.value = note;
    return o;
});

// Shape the sound by controlling its volume over time.
// Starting at time 0 quickly ramp up to full volume.
// Then starting at time 0.1 slowly ramp down to 0.
let volumeControl = audioContext.createGain();
volumeControl.gain.setTargetAtTime(1, 0.0, 0.02);
volumeControl.gain.setTargetAtTime(0, 0.1, 0.2);

// We're going to send the sound to the default destination:
// the user's speakers
let speakers = audioContext.destination;

// Connect each of the source notes to the volume control
oscillators.forEach(o => o.connect(volumeControl));

// And connect the output of the volume control to the speakers.
volumeControl.connect(speakers);

// Now start playing the sounds and let them run for 1.25 seconds.
let startTime = audioContext.currentTime;
let stopTime = startTime + 1.25;
oscillators.forEach(o => {
    o.start(startTime);
```

```
    o.stop(stopTime);
});

// If we want to create a sequence of sounds we can use event handlers
oscillators[0].addEventListener("ended", () => {
    // This event handler is invoked when the note stops playing
});
```

15.10 Location, Navigation, and History

The `location` property of both the Window and Document objects refers to the Location object, which represents the current URL of the document displayed in the window, and which also provides an API for loading new documents into the window.

The Location object is very much like a URL object (§11.9), and you can use properties like `protocol`, `hostname`, `port`, and `path` to access the various parts of the URL of the current document. The `href` property returns the entire URL as a string, as does the `toString()` method.

The `hash` and `search` properties of the Location object are interesting ones. The `hash` property returns the "fragment identifier" portion of the URL, if there is one: a hash mark (#) followed by an element ID. The `search` property is similar. It returns the portion of the URL that starts with a question mark: often some sort of query string. In general, this portion of a URL is used to parameterize the URL and provides a way to embed arguments in it. While these arguments are usually intended for scripts run on a server, there is no reason why they cannot also be used in JavaScript-enabled pages.

URL objects have a `searchParams` property that is a parsed representation of the `search` property. The Location object does not have a `searchParams` property, but if you want to parse `window.location.search`, you can simply create a URL object from the Location object and then use the URL's `searchParams`:

```
let url = new URL(window.location);
let query = url.searchParams.get("q");
let numResults = parseInt(url.searchParams.get("n") || "10");
```

In addition to the Location object that you can refer to as `window.location` or `document.location`, and the `URL()` constructor that we used earlier, browsers also define a `document.URL` property. Surprisingly, the value of this property is not a URL object, but just a string. The string holds the URL of the current document.

15.10.1 Loading New Documents

If you assign a string to `window.location` or to `document.location`, that string is interpreted as a URL and the browser loads it, replacing the current document with a new one:

```
window.location = "http://www.oreilly.com"; // Go buy some books!
```

You can also assign relative URLs to `location`. They are resolved relative to the current URL:

```
document.location = "page2.html";           // Load the next page
```

A bare fragment identifier is a special kind of relative URL that does not cause the browser to load a new document but simply to scroll so that the document element with `id` or `name` that matches the fragment is visible at the top of the browser window. As a special case, the fragment identifier `#top` makes the browser jump to the start of the document (assuming no element has an `id="top"` attribute):

```
location = "#top";                          // Jump to the top of the document
```

The individual properties of the Location object are writable, and setting them changes the location URL and also causes the browser to load a new document (or, in the case of the `hash` property, to navigate within the current document):

```
document.location.path = "pages/3.html"; // Load a new page
document.location.hash = "TOC";          // Scroll to the table of contents
location.search = "?page=" + (page+1);   // Reload with new query string
```

You can also load a new page by passing a new string to the `assign()` method of the Location object. This is the same as assigning the string to the `location` property, however, so it's not particularly interesting.

The `replace()` method of the Location object, on the other hand, is quite useful. When you pass a string to `replace()`, it is interpreted as a URL and causes the browser to load a new page, just as `assign()` does. The difference is that `replace()` replaces the current document in the browser's history. If a script in document A sets the `location` property or calls `assign()` to load document B and then the user clicks the Back button, the browser will go back to document A. If you use `replace()` instead, then document A is erased from the browser's history, and when the user clicks the Back button, the browser returns to whatever document was displayed before document A.

When a script unconditionally loads a new document, the `replace()` method is a better choice than `assign()`. Otherwise, the Back button would take the browser back to the original document, and the same script would again load the new document. Suppose you have a JavaScript-enhanced version of your page and a static version that does not use JavaScript. If you determine that the user's browser does not

support the web platform APIs that you want to use, you could use `location.replace()` to load the static version:

```
// If the browser does not support the JavaScript APIs we need,
// redirect to a static page that does not use JavaScript.
if (!isBrowserSupported()) location.replace("staticpage.html");
```

Notice that the URL passed to `replace()` is a relative one. Relative URLs are interpreted relative to the page in which they appear, just as they would be if they were used in a hyperlink.

In addition to the `assign()` and `replace()` methods, the Location object also defines `reload()`, which simply makes the browser reload the document.

15.10.2 Browsing History

The `history` property of the Window object refers to the History object for the window. The History object models the browsing history of a window as a list of documents and document states. The `length` property of the History object specifies the number of elements in the browsing history list, but for security reasons, scripts are not allowed to access the stored URLs. (If they could, any scripts could snoop through your browsing history.)

The History object has `back()` and `forward()` methods that behave like the browser's Back and Forward buttons do: they make the browser go backward or forward one step in its browsing history. A third method, `go()`, takes an integer argument and can skip any number of pages forward (for positive arguments) or backward (for negative arguments) in the history list:

```
history.go(-2);   // Go back 2, like clicking the Back button twice
history.go(0);    // Another way to reload the current page
```

If a window contains child windows (such as `<iframe>` elements), the browsing histories of the child windows are chronologically interleaved with the history of the main window. This means that calling `history.back()` (for example) on the main window may cause one of the child windows to navigate back to a previously displayed document but leaves the main window in its current state.

The History object described here dates back to the early days of the web when documents were passive and all computation was performed on the server. Today, web applications often generate or load content dynamically and display new application states without actually loading new documents. Applications like these must perform their own history management if they want the user to be able to use the Back and Forward buttons (or the equivalent gestures) to navigate from one application state to another in an intuitive way. There are two ways to accomplish this, described in the next two sections.

15.10.3 History Management with hashchange Events

One history management technique involves `location.hash` and the "hashchange" event. Here are the key facts you need to know to understand this technique:

- The `location.hash` property sets the fragment identifier of the URL and is traditionally used to specify the ID of a document section to scroll to. But `location.hash` does not have to be an element ID: you can set it to any string. As long as no element happens to have that string as its ID, the browser won't scroll when you set the `hash` property like this.
- Setting the `location.hash` property updates the URL displayed in the location bar and, very importantly, adds an entry to the browser's history.
- Whenever the fragment identifier of the document changes, the browser fires a "hashchange" event on the Window object. If you set `location.hash` explictly, a "hashchange" event is fired. And, as we've mentioned, this change to the Location object creates a new entry in the browser's browsing history. So if the user now clicks the Back button, the browser will return to its previous URL before you set `location.hash`. But this means that the fragment identifier has changed again, so another "hashchange" event is fired in this case. This means that as long as you can create a unique fragment identifier for each possible state of your application, "hashchange" events will notify you if the user moves backward and forward though their browsing history.

To use this history management mechanism, you'll need to be able to encode the state information necessary to render a "page" of your application into a relatively short string of text that is suitable for use as a fragment identifier. And you'll need to write a function to convert page state into a string and another function to parse the string and re-create the page state it represents.

Once you have written those functions, the rest is easy. Define a `window.onhashchange` function (or register a "hashchange" listener with `addEventListener()`) that reads `location.hash`, converts that string into a representation of your application state, and then takes whatever actions are necessary to display that new application state.

When the user interacts with your application (such as by clicking a link) in a way that would cause the application to enter a new state, don't render the new state directly. Instead, encode the desired new state as a string and set `location.hash` to that string. This will trigger a "hashchange" event, and your handler for that event will display the new state. Using this roundabout technique ensures that the new state is inserted into the browsing history so that the Back and Forward buttons continue to work.

15.10.4 History Management with pushState()

The second technique for managing history is somewhat more complex but is less of a hack than the "hashchange" event. This more robust history-management technique is based on the `history.pushState()` method and the "popstate" event. When a web app enters a new state, it calls `history.pushState()` to add an object representing the state to the browser's history. If the user then clicks the Back button, the browser fires a "popstate" event with a copy of that saved state object, and the app uses that object to re-create its previous state. In addition to the saved state object, applications can also save a URL with each state, which is important if you want users to be able to bookmark and share links to the internal states of the app.

The first argument to `pushState()` is an object that contains all the state information necessary to restore the current state of the document. This object is saved using HTML's *structured clone* algorithm, which is more versatile than `JSON.stringify()` and can support Map, Set, and Date objects as well as typed arrays and ArrayBuffers.

The second argument was intended to be a title string for the state, but most browsers do not support it, and you should just pass an empty string. The third argument is an optional URL that will be displayed in the location bar immediately and also if the user returns to this state via Back and Forward buttons. Relative URLs are resolved against the current location of the document. Associating a URL with each state allows the user to bookmark internal states of your application. Remember, though, that if the user saves a bookmark and then visits it a day later, you won't get a "popstate" event about that visit: you'll have to restore your application state by parsing the URL.

The Structured Clone Algorithm

The `history.pushState()` method does not use `JSON.stringify()` (§11.6) to serialize state data. Instead, it (and other browser APIs we'll learn about later) uses a more robust serialization technique known as the structured clone algorithm, defined by the HTML standard.

The structured clone algorithm can serialize anything that `JSON.stringify()` can, but in addition, it enables serialization of most other JavaScript types, including Map, Set, Date, RegExp, and typed arrays, and it can handle data structures that include circular references. The structured clone algorithm *cannot* serialize functions or classes, however. When cloning objects it does not copy the prototype object, getters and setters, or non-enumerable properties. While the structured clone algorithm can clone most built-in JavaScript types, it cannot copy types defined by the host environment, such as document Element objects.

This means that the state object you pass to `history.pushState()` need not be limited to the objects, arrays, and primitive values that `JSON.stringify()` supports. Note,

however, that if you pass an instance of a class that you have defined, that instance will be serialized as an ordinary JavaScript object and will lose its prototype.

In addition to the `pushState()` method, the History object also defines `replaceState()`, which takes the same arguments but replaces the current history state instead of adding a new state to the browsing history. When an application that uses `pushState()` is first loaded, it is often a good idea to call `replaceState()` to define a state object for this initial state of the application.

When the user navigates to saved history states using the Back or Forward buttons, the browser fires a "popstate" event on the Window object. The event object associated with the event has a property named `state`, which contains a copy (another structured clone) of the state object you passed to `pushState()`.

Example 15-9 is a simple web application—the number-guessing game pictured in Figure 15-15—that uses `pushState()` to save its history, allowing the user to "go back" to review or redo their guesses.

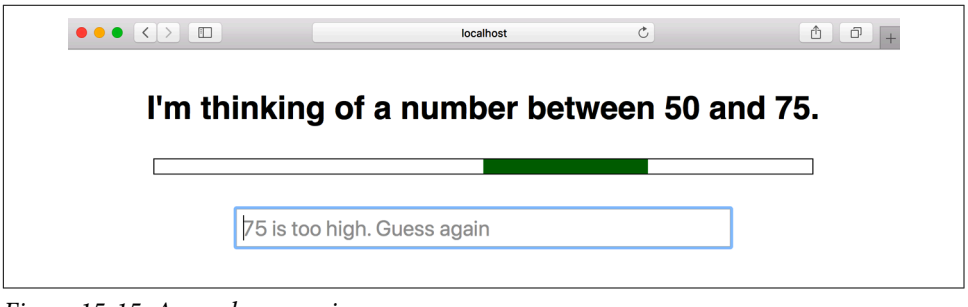

Figure 15-15. A number-guessing game

Example 15-9. History management with pushState()

```
<html><head><title>I'm thinking of a number...</title>
<style>
body { height: 250px; display: flex; flex-direction: column;
       align-items: center; justify-content: space-evenly; }
#heading { font: bold 36px sans-serif; margin: 0; }
#container { border: solid black 1px; height: 1em; width: 80%; }
#range { background-color: green; margin-left: 0%; height: 1em; width: 100%; }
#input { display: block; font-size: 24px; width: 60%; padding: 5px; }
#playagain { font-size: 24px; padding: 10px; border-radius: 5px; }
</style>
</head>
<body>
<h1 id="heading">I'm thinking of a number...</h1>
<!-- A visual representation of the numbers that have not been ruled out -->
<div id="container"><div id="range"></div></div>
```

```
<!-- Where the user enters their guess -->
<input id="input" type="text">
<!-- A button that reloads with no search string. Hidden until game ends. -->
<button id="playagain" hidden onclick="location.search='';">Play Again</button>
<script>
/**
 * An instance of this GameState class represents the internal state of
 * our number guessing game. The class defines static factory methods for
 * initializing the game state from different sources, a method for
 * updating the state based on a new guess, and a method for modifying the
 * document based on the current state.
 */
class GameState {
    // This is a factory function to create a new game
    static newGame() {
        let s = new GameState();
        s.secret = s.randomInt(0, 100);  // An integer: 0 < n < 100
        s.low = 0;                       // Guesses must be greater than this
        s.high = 100;                    // Guesses must be less than this
        s.numGuesses = 0;                // How many guesses have been made
        s.guess = null;                  // What the last guess was
        return s;
    }

    // When we save the state of the game with history.pushState(), it is just
    // a plain JavaScript object that gets saved, not an instance of GameState.
    // So this factory function re-creates a GameState object based on the
    // plain object that we get from a popstate event.
    static fromStateObject(stateObject) {
        let s = new GameState();
        for(let key of Object.keys(stateObject)) {
            s[key] = stateObject[key];
        }
        return s;
    }

    // In order to enable bookmarking, we need to be able to encode the
    // state of any game as a URL. This is easy to do with URLSearchParams.
    toURL() {
        let url = new URL(window.location);
        url.searchParams.set("l", this.low);
        url.searchParams.set("h", this.high);
        url.searchParams.set("n", this.numGuesses);
        url.searchParams.set("g", this.guess);
        // Note that we can't encode the secret number in the url or it
        // will give away the secret. If the user bookmarks the page with
        // these parameters and then returns to it, we will simply pick a
        // new random number between low and high.
        return url.href;
    }

    // This is a factory function that creates a new GameState object and
```

```
    // initializes it from the specified URL. If the URL does not contain the
    // expected parameters or if they are malformed it just returns null.
    static fromURL(url) {
        let s = new GameState();
        let params = new URL(url).searchParams;
        s.low = parseInt(params.get("l"));
        s.high = parseInt(params.get("h"));
        s.numGuesses = parseInt(params.get("n"));
        s.guess = parseInt(params.get("g"));

        // If the URL is missing any of the parameters we need or if
        // they did not parse as integers, then return null;
        if (isNaN(s.low) || isNaN(s.high) ||
            isNaN(s.numGuesses) || isNaN(s.guess)) {
            return null;
        }

        // Pick a new secret number in the right range each time we
        // restore a game from a URL.
        s.secret = s.randomInt(s.low, s.high);
        return s;
    }

    // Return an integer n, min < n < max
    randomInt(min, max) {
        return min + Math.ceil(Math.random() * (max - min - 1));
    }

    // Modify the document to display the current state of the game.
    render() {
        let heading = document.querySelector("#heading"); // The <h1> at the top
        let range = document.querySelector("#range");     // Display guess range
        let input = document.querySelector("#input");     // Guess input field
        let playagain = document.querySelector("#playagain");

        // Update the document heading and title
        heading.textContent = document.title =
            `I'm thinking of a number between ${this.low} and ${this.high}.`;

        // Update the visual range of numbers
        range.style.marginLeft = `${this.low}%`;
        range.style.width = `${(this.high-this.low)}%`;

        // Make sure the input field is empty and focused.
        input.value = "";
        input.focus();

        // Display feedback based on the user's last guess. The input
        // placeholder will show because we made the input field empty.
        if (this.guess === null) {
            input.placeholder = "Type your guess and hit Enter";
        } else if (this.guess < this.secret) {
```

```
            input.placeholder = `${this.guess} is too low. Guess again`;
        } else if (this.guess > this.secret) {
            input.placeholder = `${this.guess} is too high. Guess again`;
        } else {
            input.placeholder = document.title = `${this.guess} is correct!`;
            heading.textContent = `You win in ${this.numGuesses} guesses!`;
            playagain.hidden = false;
        }
    }

    // Update the state of the game based on what the user guessed.
    // Returns true if the state was updated, and false otherwise.
    updateForGuess(guess) {
        // If it is a number and is in the right range
        if ((guess > this.low) && (guess < this.high)) {
            // Update state object based on this guess
            if (guess < this.secret) this.low = guess;
            else if (guess > this.secret) this.high = guess;
            this.guess = guess;
            this.numGuesses++;
            return true;
        }
        else { // An invalid guess: notify user but don't update state
            alert(`Please enter a number greater than ${
                   this.low} and less than ${this.high}`);
            return false;
        }
    }
}

// With the GameState class defined, making the game work is just a matter
// of initializing, updating, saving and rendering the state object at
// the appropriate times.

// When we are first loaded, we try get the state of the game from the URL
// and if that fails we instead begin a new game. So if the user bookmarks a
// game that game can be restored from the URL. But if we load a page with
// no query parameters we'll just get a new game.
let gamestate = GameState.fromURL(window.location) || GameState.newGame();

// Save this initial state of the game into the browser history, but use
// replaceState instead of pushState() for this initial page
history.replaceState(gamestate, "", gamestate.toURL());

// Display this initial state
gamestate.render();

// When the user guesses, update the state of the game based on their guess
// then save the new state to browser history and render the new state
document.querySelector("#input").onchange = (event) => {
    if (gamestate.updateForGuess(parseInt(event.target.value))) {
        history.pushState(gamestate, "", gamestate.toURL());
```

```
    }
    gamestate.render();
};

// If the user goes back or forward in history, we'll get a popstate event
// on the window object with a copy of the state object we saved with
// pushState. When that happens, render the new state.
window.onpopstate = (event) => {
    gamestate = GameState.fromStateObject(event.state); // Restore the state
    gamestate.render();                                 // and display it
};
</script>
</body></html>
```

15.11 Networking

Every time you load a web page, the browser makes network requests—using the HTTP and HTTPS protocols—for an HTML file as well as the images, fonts, scripts, and stylesheets that the file depends on. But in addition to being able to make network requests in response to user actions, web browsers also expose JavaScript APIs for networking as well.

This section covers three network APIs:

- The `fetch()` method defines a Promise-based API for making HTTP and HTTPS requests. The `fetch()` API makes basic GET requests simple but has a comprehensive feature set that also supports just about any possible HTTP use case.
- The Server-Sent Events (or SSE) API is a convenient, event-based interface to HTTP "long polling" techniques where the web server holds the network connection open so that it can send data to the client whenever it wants.
- WebSockets is a networking protocol that is not HTTP but is designed to interoperate with HTTP. It defines an asynchronous message-passing API where clients and servers can send and receive messages from each other in a way that is similar to TCP network sockets.

15.11.1 fetch()

For basic HTTP requests, using `fetch()` is a three-step process:

1. Call `fetch()`, passing the URL whose content you want to retrieve.
2. Get the response object that is asynchronously returned by step 1 when the HTTP response begins to arrive and call a method of this response object to ask for the body of the response.

3. Get the body object that is asynchronously returned by step 2 and process it however you want.

The `fetch()` API is completely Promise-based, and there are two asynchronous steps here, so you typically expect two `then()` calls or two `await` expressions when using `fetch()`. (And if you've forgotten what those are, you may want to reread Chapter 13 before continuing with this section.)

Here's what a `fetch()` request looks like if you are using `then()` and expect the server's response to your request to be JSON-formatted:

```
fetch("/api/users/current")           // Make an HTTP (or HTTPS) GET request
    .then(response => response.json()) // Parse its body as a JSON object
    .then(currentUser => {             // Then process that parsed object
        displayUserInfo(currentUser);
    });
```

Here's a similar request made using the `async` and `await` keywords to an API that returns a plain string rather than a JSON object:

```
async function isServiceReady() {
    let response = await fetch("/api/service/status");
    let body = await response.text();
    return body === "ready";
}
```

If you understand these two code examples, then you know 80% of what you need to know to use the `fetch()` API. The subsections that follow will demonstrate how to make requests and receive responses that are somewhat more complicated than those shown here.

Goodbye XMLHttpRequest

The `fetch()` API replaces the baroque and misleadingly named XMLHttpRequest API (which has nothing to do with XML). You may still see XHR (as it is often abbreviated) in existing code, but there is no reason today to use it in new code, and it is not documented in this chapter. There is one example of XMLHttpRequest in this book, however, and you can refer to §13.1.3 if you'd like to see an example of old-style JavaScript networking.

HTTP status codes, response headers, and network errors

The three-step `fetch()` process shown in §15.11.1 elides all error-handling code. Here's a more realistic version:

```
fetch("/api/users/current")   // Make an HTTP (or HTTPS) GET request.
    .then(response => {       // When we get a response, first check it
        if (response.ok &&    // for a success code and the expected type.
```

```
            response.headers.get("Content-Type") === "application/json") {
            return response.json(); // Return a Promise for the body.
        } else {
            throw new Error(        // Or throw an error.
                `Unexpected response status ${response.status} or content type`
            );
        }
    })
    .then(currentUser => {    // When the response.json() Promise resolves
        displayUserInfo(currentUser); // do something with the parsed body.
    })
    .catch(error => {         // Or if anything went wrong, just log the error.
        // If the user's browser is offline, fetch() itself will reject.
        // If the server returns a bad response then we throw an error above.
        console.log("Error while fetching current user:", error);
    });
```

The Promise returned by `fetch()` resolves to a Response object. The `status` property of this object is the HTTP status code, such as 200 for successful requests or 404 for "Not Found" responses. (`statusText` gives the standard English text that goes along with the numeric status code.) Conveniently, the `ok` property of a Response is `true` if `status` is 200 or any code between 200 and 299 and is `false` for any other code.

`fetch()` resolves its Promise when the server's response starts to arrive, as soon as the HTTP status and response headers are available, but typically before the full response body has arrived. Even though the body is not available yet, you can examine the headers in this second step of the fetch process. The `headers` property of a Response object is a Headers object. Use its `has()` method to test for the presence of a header, or use its `get()` method to get the value of a header. HTTP header names are case-insensitive, so you can pass lowercase or mixed-case header names to these functions.

The Headers object is also iterable if you ever need to do that:

```
fetch(url).then(response => {
    for(let [name,value] of response.headers) {
        console.log(`${name}: ${value}`);
    }
});
```

If a web server responds to your `fetch()` request, then the Promise that was returned will be fulfilled with a Response object, even if the server's response was a 404 Not Found error or a 500 Internal Server Error. `fetch()` only rejects the Promise it returns if it cannot contact the web server at all. This can happen if the user's computer is offline, the server is unresponsive, or the URL specifies a hostname that does not exist. Because these things can happen on any network request, it is always a good idea to include a `.catch()` clause any time you make a `fetch()` call.

Setting request parameters

Sometimes you want to pass extra parameters along with the URL when you make a request. This can be done by adding name/value pairs at the end of a URL after a `?`. The URL and URLSearchParams classes (which were covered in §11.9) make it easy to construct URLs in this form, and the `fetch()` function accepts URL objects as its first argument, so you can include request parameters in a `fetch()` request like this:

```
async function search(term) {
    let url = new URL("/api/search");
    url.searchParams.set("q", term);
    let response = await fetch(url);
    if (!response.ok) throw new Error(response.statusText);
    let resultsArray = await response.json();
    return resultsArray;
}
```

Setting request headers

Sometimes you need to set headers in your `fetch()` requests. If you're making web API requests that require credentials, for example, then you may need to include an Authorization header that contains those credentials. In order to do this, you can use the two-argument version of `fetch()`. As before, the first argument is a string or URL object that specifies the URL to fetch. The second argument is an object that can provide additional options, including request headers:

```
let authHeaders = new Headers();
// Don't use Basic auth unless it is over an HTTPS connection.
authHeaders.set("Authorization",
                `Basic ${btoa(`${username}:${password}`)}`);
fetch("/api/users/", { headers: authHeaders })
    .then(response => response.json())            // Error handling omitted...
    .then(usersList => displayAllUsers(usersList));
```

There are a number of other options that can be specified in the second argument to `fetch()`, and we'll see it again later. An alternative to passing two arguments to `fetch()` is to instead pass the same two arguments to the `Request()` constructor and then pass the resulting Request object to `fetch()`:

```
let request = new Request(url, { headers });
fetch(request).then(response => ...);
```

Parsing response bodies

In the three-step `fetch()` process that we've demonstrated, the second step ends by calling the `json()` or `text()` methods of the Response object and returning the Promise object that those methods return. Then, the third step begins when that Promise resolves with the body of the response parsed as a JSON object or simply as a string of text.

These are probably the two most common scenarios, but they are not the only ways to obtain the body of a web server's response. In addition to `json()` and `text()`, the Response object also has these methods:

`arrayBuffer()`
: This method returns a Promise that resolves to an ArrayBuffer. This is useful when the response contains binary data. You can use the ArrayBuffer to create a typed array (§11.2) or a DataView object (§11.2.5) from which you can read the binary data.

`blob()`
: This method returns a Promise that resolves to a Blob object. Blobs are not covered in any detail in this book, but the name stands for "Binary Large Object," and they are useful when you expect large amounts of binary data. If you ask for the body of the response as a Blob, the browser implementation may stream the response data to a temporary file and then return a Blob object that represents that temporary file. Blob objects, therefore, do not allow random access to the response body the way that an ArrayBuffer does. Once you have a Blob, you can create a URL that refers to it with `URL.createObjectURL()`, or you can use the event-based FileReader API to asynchronously obtain the content of the Blob as a string or an ArrayBuffer. At the time of this writing, some browsers also define Promise-based `text()` and `arrayBuffer()` methods that give a more direct route for obtaining the content of a Blob.

`formData()`
: This method returns a Promise that resolves to a FormData object. You should use this method if you expect the body of the Response to be encoded in "multipart/form-data" format. This format is common in POST requests made to a server, but uncommon in server responses, so this method is not frequently used.

Streaming response bodies

In addition to the five response methods that asynchronously return some form of the complete response body to you, there is also an option to stream the response body, which is useful if there is some kind of processing you can do on the chunks of the response body as they arrive over the network. But streaming the response is also useful if you want to display a progress bar so that the user can see the progress of the download.

The `body` property of a Response object is a ReadableStream object. If you have already called a response method like `text()` or `json()` that reads, parses, and returns the body, then `bodyUsed` will be `true` to indicate that the `body` stream has already been read. If `bodyUsed` is `false`, however, then the stream has not yet been read. In this case, you can call `getReader()` on `response.body` to obtain a stream

reader object, then use the `read()` method of this reader object to asynchronously read chunks of text from the stream. The `read()` method returns a Promise that resolves to an object with `done` and `value` properties. `done` will be `true` if the entire body has been read or if the stream was closed. And `value` will either be the next chunk, as a Uint8Array, or `undefined` if there are no more chunks.

This streaming API is relatively straightforward if you use `async` and `await` but is surprisingly complex if you attempt to use it with raw Promises. Example 15-10 demonstrates the API by defining a `streamBody()` function. Suppose you wanted to download a large JSON file and report download progress to the user. You can't do that with the `json()` method of the Response object, but you could do it with the `streamBody()` function, like this (assuming that an `updateProgress()` function is defined to set the `value` attribute on an HTML `<progress>` element):

```
fetch('big.json')
    .then(response => streamBody(response, updateProgress))
    .then(bodyText => JSON.parse(bodyText))
    .then(handleBigJSONObject);
```

The `streamBody()` function can be implemented as shown in Example 15-10.

Example 15-10. Streaming the response body from a fetch() request

```
/**
 * An asynchronous function for streaming the body of a Response object
 * obtained from a fetch() request. Pass the Response object as the first
 * argument followed by two optional callbacks.
 *
 * If you specify a function as the second argument, that reportProgress
 * callback will be called once for each chunk that is received. The first
 * argument passed is the total number of bytes received so far. The second
 * argument is a number between 0 and 1 specifying how complete the download
 * is. If the Response object has no "Content-Length" header, however, then
 * this second argument will always be NaN.
 *
 * If you want to process the data in chunks as they arrive, specify a
 * function as the third argument. The chunks will be passed, as Uint8Array
 * objects, to this processChunk callback.
 *
 * streamBody() returns a Promise that resolves to a string. If a processChunk
 * callback was supplied then this string is the concatenation of the values
 * returned by that callback. Otherwise the string is the concatenation of
 * the chunk values converted to UTF-8 strings.
 */
async function streamBody(response, reportProgress, processChunk) {
    // How many bytes are we expecting, or NaN if no header
    let expectedBytes = parseInt(response.headers.get("Content-Length"));
    let bytesRead = 0;                      // How many bytes received so far
    let reader = response.body.getReader();  // Read bytes with this function
```

```
    let decoder = new TextDecoder("utf-8");  // For converting bytes to text
    let body = "";                           // Text read so far

    while(true) {                                    // Loop until we exit below
        let {done, value} = await reader.read();  // Read a chunk

        if (value) {                                 // If we got a byte array:
            if (processChunk) {                      // Process the bytes if
                let processed = processChunk(value);  // a callback was passed.
                if (processed) {
                    body += processed;
                }
            } else {                                 // Otherwise, convert bytes
                body += decoder.decode(value, {stream: true}); // to text.
            }

            if (reportProgress) {                    // If a progress callback was
                bytesRead += value.length;           // passed, then call it
                reportProgress(bytesRead, bytesRead / expectedBytes);
            }
        }
        if (done) {                                  // If this is the last chunk,
            break;                                   // exit the loop
        }
    }

    return body;   // Return the body text we accumulated
}
```

This streaming API is new at the time of this writing and is expected to evolve. In particular, there are plans to make ReadableStream objects asynchronously iterable so that they can be used with `for/await` loops (§13.4.1).

Specifying the request method and request body

In each of the `fetch()` examples shown so far, we have made an HTTP (or HTTPS) GET request. If you want to use a different request method (such as POST, PUT, or DELETE), simply use the two-argument version of `fetch()`, passing an Options object with a `method` parameter:

```
fetch(url, { method: "POST" }).then(r => r.json()).then(handleResponse);
```

POST and PUT requests typically have a request body containing data to be sent to the server. As long as the `method` property is not set to `"GET"` or `"HEAD"` (which do not support request bodies), you can specify a request body by setting the `body` property of the Options object:

```
fetch(url, {
    method: "POST",
    body: "hello world"
})
```

When you specify a request body, the browser automatically adds an appropriate "Content-Length" header to the request. When the body is a string, as in the preceding example, the browser defaults the "Content-Type" header to "text/plain;charset=UTF-8." You may need to override this default if you specify a string body of some more specific type such as "text/html" or "application/json":

```
fetch(url, {
    method: "POST",
    headers: new Headers({"Content-Type": "application/json"}),
    body: JSON.stringify(requestBody)
})
```

The `body` property of the `fetch()` options object does not have to be a string. If you have binary data in a typed array or a DataView object or an ArrayBuffer, you can set the `body` property to that value and specify an appropriate "Content-Type" header. If you have binary data in Blob form, you can simply set `body` to the Blob. Blobs have a `type` property that specifies their content type, and the value of this property is used as the default value of the "Content-Type" header.

With POST requests, is it somewhat common to pass a set of name/value parameters in the request body (instead of encoding them into the query portion of the URL). There are two ways to do this:

- You can specify your parameter names and values with URLSearchParams (which we saw earlier in this section, and which is documented in §11.9) and then pass the URLSearchParams object as the value of the `body` property. If you do this, the body will be set to a string that looks like the query portion of a URL, and the "Content-Type" header will be automatically set to "application/x-www-form-urlencoded;charset=UTF-8."
- If instead you specify your parameter names and values with a FormData object, the body will use a more verbose multipart encoding and "Content-Type" will be set to "multipart/form-data; boundary=…" with a unique boundary string that matches the body. Using a FormData object is particularly useful when the values you want to upload are long, or are File or Blob objects that may each have its own "Content-Type." FormData objects can be created and initialized with values by passing a `<form>` element to the `FormData()` constructor. But you can also create "multipart/form-data" request bodies by invoking the `FormData()` constructor with no arguments and initializing the name/value pairs it represents with the `set()` and `append()` methods.

File upload with fetch()

Uploading files from a user's computer to a web server is a common task and can be accomplished using a FormData object as the request body. A common way to obtain a File object is to display an `<input type="file">` element on your web page and listen for "change" events on that element. When a "change" event occurs, the `files` array of the input element should contain at least one File object. File objects are also available through the HTML drag-and-drop API. That API is not covered in this book, but you can get files from the `dataTransfer.files` array of the event object passed to an event listener for "drop" events.

Remember also that File objects are a kind of Blob, and sometimes it can be useful to upload Blobs. Suppose you've written a web application that allows the user to create drawings in a `<canvas>` element. You can upload the user's drawings as PNG files with code like the following:

```
// The canvas.toBlob() function is callback-based.
// This is a Promise-based wrapper for it.
async function getCanvasBlob(canvas) {
    return new Promise((resolve, reject) => {
        canvas.toBlob(resolve);
    });
}

// Here is how we upload a PNG file from a canvas
async function uploadCanvasImage(canvas) {
    let pngblob = await getCanvasBlob(canvas);
    let formdata = new FormData();
    formdata.set("canvasimage", pngblob);
    let response = await fetch("/upload", { method: "POST", body: formdata });
    let body = await response.json();
}
```

Cross-origin requests

Most often, `fetch()` is used by web applications to request data from their own web server. Requests like these are known as same-origin requests because the URL passed to `fetch()` has the same origin (protocol plus hostname plus port) as the document that contains the script that is making the request.

For security reasons, web browsers generally disallow (though there are exceptions for images and scripts) cross-origin network requests. However, Cross-Origin Resource Sharing, or CORS, enables safe cross-origin requests. When `fetch()` is used with a cross-origin URL, the browser adds an "Origin" header to the request (and does not allow it to be overridden via the `headers` property) to notify the web server that the request is coming from a document with a different origin. If the server responds to the request with an appropriate "Access-Control-Allow-Origin" header,

then the request proceeds. Otherwise, if the server does not explicitly allow the request, then the Promise returned by `fetch()` is rejected.

Aborting a request

Sometimes you may want to abort a `fetch()` request that you have already issued, perhaps because the user clicked a Cancel button or the request is taking too long. The fetch API allows requests to be aborted using the AbortController and AbortSignal classes. (These classes define a generic abort mechanism suitable for use by other APIs as well.)

If you want to have the option of aborting a `fetch()` request, then create an AbortController object before starting the request. The `signal` property of the controller object is an AbortSignal object. Pass this signal object as the value of the `signal` property of the options object that you pass to `fetch()`. Having done that, you can call the `abort()` method of the controller object to abort the request, which will cause any Promise objects related to the fetch request to reject with an exception.

Here is an example of using the AbortController mechanism to enforce a timeout for fetch requests:

```
// This function is like fetch(), but it adds support for a timeout
// property in the options object and aborts the fetch if it is not complete
// within the number of milliseconds specified by that property.
function fetchWithTimeout(url, options={}) {
    if (options.timeout) {  // If the timeout property exists and is nonzero
        let controller = new AbortController();  // Create a controller
        options.signal = controller.signal;      // Set the signal property
        // Start a timer that will send the abort signal after the specified
        // number of milliseconds have passed. Note that we never cancel
        // this timer. Calling abort() after the fetch is complete has
        // no effect.
        setTimeout(() => { controller.abort(); }, options.timeout);
    }
    // Now just perform a normal fetch
    return fetch(url, options);
}
```

Miscellaneous request options

We've seen that an Options object can be passed as the second argument to `fetch()` (or as the second argument to the `Request()` constructor) to specify the request method, request headers, and request body. It supports a number of other options as well, including these:

`cache`
: Use this property to override the browser's default caching behavior. HTTP caching is a complex topic that is beyond the scope of this book, but if you know something about how it works, you can use the following legal values of `cache`:

`"default"`
: This value specifies the default caching behavior. Fresh responses in the cache are served directly from the cache, and stale responses are revalidated before being served.

`"no-store"`
: This value makes the browser ignore its cache. The cache is not checked for matches when the request is made and is not updated when the response arrives.

`"reload"`
: This value tells the browser to always make a normal network request, ignoring the cache. When the response arrives, however, it is stored in the cache.

`"no-cache"`
: This (misleadingly named) value tells the browser to not serve fresh values from the cache. Fresh or stale cached values are revalidated before being returned.

`"force-cache"`
: This value tells the browser to serve responses from the cache even if they are stale.

`redirect`
: This property controls how the browser handles redirect responses from the server. The three legal values are:

`"follow"`
: This is the default value, and it makes the browser follow redirects automatically. If you use this default, the Response objects you get with `fetch()` should never have a `status` in the 300 to 399 range.

`"error"`
: This value makes `fetch()` reject its returned Promise if the server returns a redirect response.

`"manual"`
: This value means that you want to manually handle redirect responses, and the Promise returned by `fetch()` may resolve to a Response object with a `status` in the 300 to 399 range. In this case, you will have to use the "Location" header of the Response to manually follow the redirection.

`referrer`
: You can set this property to a string that contains a relative URL to specify the value of the HTTP "Referer" header (which is historically misspelled with three Rs instead of four). If you set this property to the empty string, then the "Referer" header will be omitted from the request.

15.11.2 Server-Sent Events

A fundamental feature of the HTTP protocol upon which the web is built is that clients initiate requests and servers respond to those requests. Some web apps find it useful, however, to have their server send them notifications when events occur. This does not come naturally to HTTP, but the technique that has been devised is for the client to make a request to the server, and then neither the client nor the server close the connection. When the server has something to tell the client about, it writes data to the connection but keeps it open. The effect is as if the client makes a network request and the server responds in a slow and bursty way with significant pauses between bursts of activity. Network connections like this don't usually stay open forever, but if the client detects that the connection has closed, it can simply make another request to reopen the connection.

This technique for allowing servers to send messages to clients is surprisingly effective (though it can be expensive on the server side because the server must maintain an active connection to all of its clients). Because it is a useful programming pattern, client-side JavaScript supports it with the EventSource API. To create this kind of long-lived request connection to a web server, simply pass a URL to the `EventSource()` constructor. When the server writes (properly formatted) data to the connection, the EventSource object translates those into events that you can listen for:

```
let ticker = new EventSource("stockprices.php");
ticker.addEventListener("bid", (event) => {
    displayNewBid(event.data);
}
```

The event object associated with a message event has a `data` property that holds whatever string the server sent as the payload for this event. The event object also has a `type` property, like all event objects do, that specifies the name of the event. The server determines the type of the events that are generated. If the server omits an event name in the data it writes, then the event type defaults to "message."

The Server-Sent Event protocol is straightforward. The client initiates a connection to the server (when it creates the `EventSource` object), and the server keeps this connection open. When an event occurs, the server writes lines of text to the connection. An event going over the wire might look like this, if the comments were omitted:

```
event: bid  // sets the type of the event object
data: GOOG  // sets the data property
```

```
    data: 999   // appends a newline and more data
                // a blank line marks the end of the event
```

There are some additional details to the protocol that allow events to be given IDs and allow a reconnecting client to tell the server what the ID of the last event it received was, so that a server can resend any events it missed. Those details are invisible on the client side, however, and are not discussed here.

One obvious application for Server-Sent Events is for multiuser collaborations like online chat. A chat client might use `fetch()` to post messages to the chat room and subscribe to the stream of chatter with an EventSource object. Example 15-11 demonstrates how easy it is to write a chat client like this with EventSource.

Example 15-11. A simple chat client using EventSource

```
<html>
<head><title>SSE Chat</title></head>
<body>
<!-- The chat UI is just a single text input field -->
<!-- New chat messages will be inserted before this input field -->
<input id="input" style="width:100%; padding:10px; border:solid black 2px"/>
<script>
// Take care of some UI details
let nick = prompt("Enter your nickname");     // Get user's nickname
let input = document.getElementById("input"); // Find the input field
input.focus();                                // Set keyboard focus

// Register for notification of new messages using EventSource
let chat = new EventSource("/chat");
chat.addEventListener("chat", event => {  // When a chat message arrives
    let div = document.createElement("div");  // Create a <div>
    div.append(event.data);                   // Add text from the message
    input.before(div);                        // And add div before input
    input.scrollIntoView();                   // Ensure input elt is visible
});

// Post the user's messages to the server using fetch
input.addEventListener("change", ()=>{  // When the user strikes return
    fetch("/chat", {                    // Start an HTTP request to this url.
        method: "POST",                 // Make it a POST request with body
        body: nick + ": " + input.value // set to the user's nick and input.
    })
    .catch(e => console.error);         // Ignore response, but log any errors.
    input.value = "";                   // Clear the input
});
</script>
</body>
</html>
```

The server-side code for this chat program is not much more complicated than the client-side code. Example 15-12 is a simple Node HTTP server. When a client requests the root URL "/", it sends the chat client code shown in Example 15-11. When a client makes a GET request for the URL "/chat", it saves the response object and keeps that connection open. And when a client makes a POST request to "/chat", it uses the body of the request as a chat message and writes it, using the "text/event-stream" format to each of the saved response objects. The server code listens on port 8080, so after running it with Node, point your browser to `http://localhost:8080` to connect and begin chatting with yourself.

Example 15-12. A Server-Sent Events chat server

```
// This is server-side JavaScript, intended to be run with NodeJS.
// It implements a very simple, completely anonymous chat room.
// POST new messages to /chat, or GET a text/event-stream of messages
// from the same URL. Making a GET request to / returns a simple HTML file
// that contains the client-side chat UI.
const http = require("http");
const fs = require("fs");
const url = require("url");

// The HTML file for the chat client. Used below.
const clientHTML = fs.readFileSync("chatClient.html");

// An array of ServerResponse objects that we're going to send events to
let clients = [];

// Create a new server, and listen on port 8080.
// Connect to http://localhost:8080/ to use it.
let server = new http.Server();
server.listen(8080);

// When the server gets a new request, run this function
server.on("request", (request, response) => {
    // Parse the requested URL
    let pathname = url.parse(request.url).pathname;

    // If the request was for "/", send the client-side chat UI.
    if (pathname === "/") {  // A request for the chat UI
        response.writeHead(200, {"Content-Type": "text/html"}).end(clientHTML);
    }
    // Otherwise send a 404 error for any path other than "/chat" or for
    // any method other than "GET" and "POST"
    else if (pathname !== "/chat" ||
             (request.method !== "GET" && request.method !== "POST")) {
        response.writeHead(404).end();
    }
    // If the /chat request was a GET, then a client is connecting.
    else if (request.method === "GET") {
```

```
        acceptNewClient(request, response);
    }
    // Otherwise the /chat request is a POST of a new message
    else {
        broadcastNewMessage(request, response);
    }
});

// This handles GET requests for the /chat endpoint which are generated when
// the client creates a new EventSource object (or when the EventSource
// reconnects automatically).
function acceptNewClient(request, response) {
    // Remember the response object so we can send future messages to it
    clients.push(response);

    // If the client closes the connection, remove the corresponding
    // response object from the array of active clients
    request.connection.on("end", () => {
        clients.splice(clients.indexOf(response), 1);
        response.end();
    });

    // Set headers and send an initial chat event to just this one client
    response.writeHead(200, {
        "Content-Type": "text/event-stream",
        "Connection": "keep-alive",
        "Cache-Control": "no-cache"
    });
    response.write("event: chat\ndata: Connected\n\n");

    // Note that we intentionally do not call response.end() here.
    // Keeping the connection open is what makes Server-Sent Events work.
}

// This function is called in response to POST requests to the /chat endpoint
// which clients send when users type a new message.
async function broadcastNewMessage(request, response) {
    // First, read the body of the request to get the user's message
    request.setEncoding("utf8");
    let body = "";
    for await (let chunk of request) {
        body += chunk;
    }

    // Once we've read the body send an empty response and close the connection
    response.writeHead(200).end();

    // Format the message in text/event-stream format, prefixing each
    // line with "data: "
    let message = "data: " + body.replace("\n", "\ndata: ");

    // Give the message data a prefix that defines it as a "chat" event
```

```
    // and give it a double newline suffix that marks the end of the event.
    let event = `event: chat\n${message}\n\n`;

    // Now send this event to all listening clients
    clients.forEach(client => client.write(event));
}
```

15.11.3 WebSockets

The WebSocket API is a simple interface to a complex and powerful network protocol. WebSockets allow JavaScript code in the browser to easily exchange text and binary messages with a server. As with Server-Sent Events, the client must establish the connection, but once the connection is established, the server can asynchronously send messages to the client. Unlike SSE, binary messages are supported, and messages can be sent in both directions, not just from server to client.

The network protocol that enables WebSockets is a kind of extension to HTTP. Although the WebSocket API is reminiscent of low-level network sockets, connection endpoints are not identified by IP address and port. Instead, when you want to connect to a service using the WebSocket protocol, you specify the service with a URL, just as you would for a web service. WebSocket URLs begin with `wss://` instead of `https://`, however. (Browsers typically restrict WebSockets to only work in pages loaded over secure `https://` connections).

To establish a WebSocket connection, the browser first establishes an HTTP connection and sends the server an `Upgrade: websocket` header requesting that the connection be switched from the HTTP protocol to the WebSocket protocol. What this means is that in order to use WebSockets in your client-side JavaScript, you will need to be working with a web server that also speaks the WebSocket protocol, and you will need to have server-side code written to send and receive data using that protocol. If your server is set up that way, then this section will explain everything you need to know to handle the client-side end of the connection. If your server does not support the WebSocket protocol, consider using Server-Sent Events (§15.11.2) instead.

Creating, connecting, and disconnecting WebSockets

If you want to communicate with a WebSocket-enabled server, create a WebSocket object, specifying the `wss://` URL that identifies the server and service you want to use:

```
let socket = new WebSocket("wss://example.com/stockticker");
```

When you create a WebSocket, the connection process begins automatically. But a newly created WebSocket will not be connected when it is first returned.

The `readyState` property of the socket specifies what state the connection is in. This property can have the following values:

`WebSocket.CONNECTING`
: This WebSocket is connecting.

`WebSocket.OPEN`
: This WebSocket is connected and ready for communication.

`WebSocket.CLOSING`
: This WebSocket connection is being closed.

`WebSocket.CLOSED`
: This WebSocket has been closed; no further communication is possible. This state can also occur when the initial connection attempt fails.

When a WebSocket transitions from the CONNECTING to the OPEN state, it fires an "open" event, and you can listen for this event by setting the `onopen` property of the WebSocket or by calling `addEventListener()` on that object.

If a protocol or other error occurs for a WebSocket connection, the WebSocket object fires an "error" event. You can set `onerror` to define a handler, or, alternatively, use `addEventListener()`.

When you are done with a WebSocket, you can close the connection by calling the `close()` method of the WebSocket object. When a WebSocket changes to the CLOSED state, it fires a "close" event, and you can set the `onclose` property to listen for this event.

Sending messages over a WebSocket

To send a message to the server on the other end of a WebSocket connection, simply invoke the `send()` method of the WebSocket object. `send()` expects a single message argument, which can be a string, Blob, ArrayBuffer, typed array, or DataView object.

The `send()` method buffers the specified message to be transmitted and returns before the message is actually sent. The `bufferedAmount` property of the WebSocket object specifies the number of bytes that are buffered but not yet sent. (Surprisingly, WebSockets do not fire any event when this value reaches 0.)

Receiving messages from a WebSocket

To receive messages from a server over a WebSocket, register an event handler for "message" events, either by setting the `onmessage` property of the WebSocket object, or by calling `addEventListener()`. The object associated with a "message" event is a MessageEvent instance with a `data` property that contains the server's message. If the server sent UTF-8 encoded text, then `event.data` will be a string containing that text.

If the server sends a message that consists of binary data instead of text, then the `data` property will (by default) be a Blob object representing that data. If you prefer to receive binary messages as ArrayBuffers instead of Blobs, set the `binaryType` property of the WebSocket object to the string "arraybuffer."

There are a number of Web APIs that use MessageEvent objects for exchanging messages. Some of these APIs use the structured clone algorithm (see "The Structured Clone Algorithm" on page 513) to allow complex data structures as the message payload. WebSockets is not one of those APIs: messages exchanged over a WebSocket are either a single string of Unicode characters or a single string of bytes (represented as a Blob or an ArrayBuffer).

Protocol negotiation

The WebSocket protocol enables the exchange of text and binary messages, but says nothing at all about the structure or meaning of those messages. Applications that use WebSockets must build their own communication protocol on top of this simple message-exchange mechanism. The use of `wss://` URLs helps with this: each URL will typically have its own rules for how messages are to be exchanged. If you write code to connect to `wss://example.com/stockticker`, then you probably know that you will be receiving messages about stock prices.

Protocols tend to evolve, however. If a hypothetical stock quotation protocol is updated, you can define a new URL and connect to the updated service as `wss://example.com/stockticker/v2`. URL-based versioning is not always sufficient, however. With complex protocols that have evolved over time, you may end up with deployed servers that support multiple versions of the protocol and deployed clients that support a different set of protocol versions.

Anticipating this situation, the WebSocket protocol and API include an application-level protocol negotiation feature. When you call the `WebSocket()` constructor, the `wss://` URL is the first argument, but you can also pass an array of strings as the second argument. If you do this, you are specifying a list of application protocols that you know how to handle and asking the server to pick one. During the connection process, the server will choose one of the protocols (or will fail with an error if it does not support any of the client's options). Once the connection has been established, the `protocol` property of the WebSocket object specifies which protocol version the server chose.

15.12 Storage

Web applications can use browser APIs to store data locally on the user's computer. This client-side storage serves to give the web browser a memory. Web apps can store user preferences, for example, or even store their complete state, so that they can resume exactly where you left off at the end of your last visit. Client-side storage is segregated by origin, so pages from one site can't read the data stored by pages from another site. But two pages from the same site can share storage and use it as a communication mechanism. Data input in a form on one page can be displayed in a table on another page, for example. Web applications can choose the lifetime of the data they store: data can be stored temporarily so that it is retained only until the window closes or the browser exits, or it can be saved on the user's computer and stored permanently so that it is available months or years later.

There are a number of forms of client-side storage:

Web Storage
: The Web Storage API consists of the `localStorage` and `sessionStorage` objects, which are essentially persistent objects that map string keys to string values. Web Storage is very easy to use and is suitable for storing large (but not huge) amounts of data.

Cookies
: Cookies are an old client-side storage mechanism that was designed for use by server-side scripts. An awkward JavaScript API makes cookies scriptable on the client side, but they're hard to use and suitable only for storing small amounts of textual data. Also, any data stored as cookies is always transmitted to the server with every HTTP request, even if the data is only of interest to the client.

IndexedDB
: IndexedDB is an asynchronous API to an object database that supports indexing.

Storage, Security, and Privacy

Web browsers often offer to remember web passwords for you, and they store them safely in encrypted form on the device. But none of the forms of client-side data storage described in this chapter involve encryption: you should assume that anything your web applications save resides on the user's device in unencrypted form. Stored data is therefore accessible to curious users who share access to the device and to malicious software (such as spyware) that exists on the device. For this reason, no form of client-side storage should ever be used for passwords, financial account numbers, or other similarly sensitive information.

15.12.1 localStorage and sessionStorage

The `localStorage` and `sessionStorage` properties of the Window object refer to Storage objects. A Storage object behaves much like a regular JavaScript object, except that:

- The property values of Storage objects must be strings.
- The properties stored in a Storage object persist. If you set a property of the localStorage object and then the user reloads the page, the value you saved in that property is still available to your program.

You can use the localStorage object like this, for example:

```
let name = localStorage.username;         // Query a stored value.
if (!name) {
    name = prompt("What is your name?"); // Ask the user a question.
    localStorage.username = name;         // Store the user's response.
}
```

You can use the `delete` operator to remove properties from `localStorage` and `sessionStorage`, and you can use a `for/in` loop or `Object.keys()` to enumerate the properties of a Storage object. If you want to remove all properties of a storage object, call the `clear()` method:

```
localStorage.clear();
```

Storage objects also define `getItem()`, `setItem()`, and `deleteItem()` methods, which you can use instead of direct property access and the `delete` operator if you want to.

Keep in mind that the properties of Storage objects can only store strings. If you want to store and retrieve other kinds of data, you'll have to encode and decode it yourself.

For example:

```
// If you store a number, it is automatically converted to a string.
// Don't forget to parse it when retrieving it from storage.
localStorage.x = 10;
let x = parseInt(localStorage.x);

// Convert a Date to a string when setting, and parse it when getting
localStorage.lastRead = (new Date()).toUTCString();
let lastRead = new Date(Date.parse(localStorage.lastRead));

// JSON makes a convenient encoding for any primitive or data structure
localStorage.data = JSON.stringify(data);  // Encode and store
let data = JSON.parse(localStorage.data);  // Retrieve and decode.
```

Storage lifetime and scope

The difference between `localStorage` and `sessionStorage` involves the lifetime and scope of the storage. Data stored through `localStorage` is permanent: it does not expire and remains stored on the user's device until a web app deletes it or the user asks the browser (through some browser-specific UI) to delete it.

`localStorage` is scoped to the document origin. As explained in "The same-origin policy" on page 424, the origin of a document is defined by its protocol, hostname, and port. All documents with the same origin share the same `localStorage` data (regardless of the origin of the scripts that actually access `localStorage`). They can read each other's data, and they can overwrite each other's data. But documents with different origins can never read or overwrite each other's data (even if they're both running a script from the same third-party server).

Note that `localStorage` is also scoped by browser implementation. If you visit a site using Firefox and then visit again using Chrome (for example), any data stored during the first visit will not be accessible during the second visit.

Data stored through `sessionStorage` has a different lifetime than data stored through `localStorage`: it has the same lifetime as the top-level window or browser tab in which the script that stored it is running. When the window or tab is permanently closed, any data stored through `sessionStorage` is deleted. (Note, however, that modern browsers have the ability to reopen recently closed tabs and restore the last browsing session, so the lifetime of these tabs and their associated `sessionStorage` may be longer than it seems.)

Like `localStorage`, `sessionStorage` is scoped to the document origin so that documents with different origins will never share `sessionStorage`. But `sessionStorage` is also scoped on a per-window basis. If a user has two browser tabs displaying documents from the same origin, those two tabs have separate `sessionStorage` data: the scripts running in one tab cannot read or overwrite the data written by scripts in the other tab, even if both tabs are visiting exactly the same page and are running exactly the same scripts.

Storage events

Whenever the data stored in `localStorage` changes, the browser triggers a "storage" event on any other Window objects to which that data is visible (but not on the window that made the change). If a browser has two tabs open to pages with the same origin, and one of those pages stores a value in `localStorage`, the other tab will receive a "storage" event.

Register a handler for "storage" events either by setting `window.onstorage` or by calling `window.addEventListener()` with event type "storage".

The event object associated with a "storage" event has some important properties:

`key`
: The name or key of the item that was set or removed. If the `clear()` method was called, this property will be `null`.

`newValue`
: Holds the new value of the item, if there is one. If `removeItem()` was called, this property will not be present.

`oldValue`
: Holds the old value of an existing item that changed or was deleted. If a new property (with no old value) is added, then this property will not be present in the event object.

`storageArea`
: The Storage object that changed. This is usually the `localStorage` object.

`url`
: The URL (as a string) of the document whose script made this storage change.

Note that `localStorage` and the "storage" event can serve as a broadcast mechanism by which a browser sends a message to all windows that are currently visiting the same website. If a user requests that a website stop performing animations, for example, the site might store that preference in `localStorage` so that it can honor it in future visits. And by storing the preference, it generates an event that allows other windows displaying the same site to honor the request as well.

As another example, imagine a web-based image-editing application that allows the user to display tool palettes in separate windows. When the user selects a tool, the application uses `localStorage` to save the current state and to generate a notification to other windows that a new tool has been selected.

15.12.2 Cookies

A *cookie* is a small amount of named data stored by the web browser and associated with a particular web page or website. Cookies were designed for server-side programming, and at the lowest level, they are implemented as an extension to the HTTP protocol. Cookie data is automatically transmitted between the web browser and web server, so server-side scripts can read and write cookie values that are stored on the client. This section demonstrates how client-side scripts can also manipulate cookies using the `cookie` property of the Document object.

Why "Cookie"?

The name "cookie" does not have a lot of significance, but it is not used without precedent. In the annals of computing history, the term "cookie" or "magic cookie" has been used to refer to a small chunk of data, particularly a chunk of privileged or secret data, akin to a password, that proves identity or permits access. In JavaScript, cookies are used to save state and can establish a kind of identity for a web browser. Cookies in JavaScript do not use any kind of cryptography, however, and are not secure in any way (although transmitting them across an `https:` connection helps).

The API for manipulating cookies is an old and cryptic one. There are no methods involved: cookies are queried, set, and deleted by reading and writing the `cookie` property of the Document object using specially formatted strings. The lifetime and scope of each cookie can be individually specified with cookie attributes. These attributes are also specified with specially formatted strings set on the same `cookie` property.

The subsections that follow explain how to query and set cookie values and attributes.

Reading cookies

When you read the `document.cookie` property, it returns a string that contains all the cookies that apply to the current document. The string is a list of name/value pairs separated from each other by a semicolon and a space. The cookie value is just the value itself and does not include any of the attributes that may be associated with that cookie. (We'll talk about attributes next.) In order to make use of the `document.cookie` property, you must typically call the `split()` method to break it into individual name/value pairs.

Once you have extracted the value of a cookie from the `cookie` property, you must interpret that value based on whatever format or encoding was used by the cookie's creator. You might, for example, pass the cookie value to `decodeURIComponent()` and then to `JSON.parse()`.

The code that follows defines a `getCookie()` function that parses the `document.cookie` property and returns an object whose properties specify the names and values of the document's cookies:

```
// Return the document's cookies as a Map object.
// Assume that cookie values are encoded with encodeURIComponent().
function getCookies() {
    let cookies = new Map();    // The object we will return
    let all = document.cookie;  // Get all cookies in one big string
    let list = all.split("; "); // Split into individual name/value pairs
    for(let cookie of list) {   // For each cookie in that list
```

```
            if (!cookie.includes("=")) continue; // Skip if there is no = sign
            let p = cookie.indexOf("=");         // Find the first = sign
            let name = cookie.substring(0, p);   // Get cookie name
            let value = cookie.substring(p+1);   // Get cookie value
            value = decodeURIComponent(value);   // Decode the value
            cookies.set(name, value);            // Remember cookie name and value
        }
        return cookies;
    }
```

Cookie attributes: lifetime and scope

In addition to a name and a value, each cookie has optional attributes that control its lifetime and scope. Before we can describe how to set cookies with JavaScript, we need to explain cookie attributes.

Cookies are transient by default; the values they store last for the duration of the web browser session but are lost when the user exits the browser. If you want a cookie to last beyond a single browsing session, you must tell the browser how long (in seconds) you would like it to retain the cookie by specifying a `max-age` attribute. If you specify a lifetime, the browser will store cookies in a file and delete them only once they expire.

Cookie visibility is scoped by document origin as `localStorage` and `sessionStorage` are, but also by document path. This scope is configurable through cookie attributes `path` and `domain`. By default, a cookie is associated with, and accessible to, the web page that created it and any other web pages in the same directory or any subdirectories of that directory. If the web page *example.com/catalog/index.html* creates a cookie, for example, that cookie is also visible to *example.com/catalog/order.html* and *example.com/catalog/widgets/index.html*, but it is not visible to *example.com/about.html*.

This default visibility behavior is often exactly what you want. Sometimes, though, you'll want to use cookie values throughout a website, regardless of which page creates the cookie. For instance, if the user enters their mailing address in a form on one page, you may want to save that address to use as the default the next time they return to the page and also as the default in an entirely unrelated form on another page where they are asked to enter a billing address. To allow this usage, you specify a `path` for the cookie. Then, any web page from the same web server whose URL begins with the path prefix you specified can share the cookie. For example, if a cookie set by *example.com/catalog/widgets/index.html* has its path set to "/catalog", that cookie is also visible to *example.com/catalog/order.html*. Or, if the path is set to "/", the cookie is visible to any page in the *example.com* domain, giving the cookie a scope like that of `localStorage`.

By default, cookies are scoped by document origin. Large websites may want cookies to be shared across subdomains, however. For example, the server at

order.example.com may need to read cookie values set from *catalog.example.com*. This is where the `domain` attribute comes in. If a cookie created by a page on *catalog.example.com* sets its `path` attribute to "/" and its `domain` attribute to ".example.com," that cookie is available to all web pages on *catalog.example.com*, *orders.example.com*, and any other server in the *example.com* domain. Note that you cannot set the domain of a cookie to a domain other than a parent domain of your server.

The final cookie attribute is a boolean attribute named `secure` that specifies how cookie values are transmitted over the network. By default, cookies are insecure, which means that they are transmitted over a normal, insecure HTTP connection. If a cookie is marked secure, however, it is transmitted only when the browser and server are connected via HTTPS or another secure protocol.

Cookie Limitations

Cookies are intended for storage of small amounts of data by server-side scripts, and that data is transferred to the server each time a relevant URL is requested. The standard that defines cookies encourages browser manufacturers to allow unlimited numbers of cookies of unrestricted size but does not require browsers to retain more than 300 cookies total, 20 cookies per web server, or 4 KB of data per cookie (both name and value count toward this 4 KB limit). In practice, browsers allow many more than 300 cookies total, but the 4 KB size limit may still be enforced by some.

Storing cookies

To associate a transient cookie value with the current document, simply set the `cookie` property to a `name=value` string. For example:

```
document.cookie = `version=${encodeURIComponent(document.lastModified)}`;
```

The next time you read the `cookie` property, the name/value pair you stored is included in the list of cookies for the document. Cookie values cannot include semicolons, commas, or whitespace. For this reason, you may want to use the core JavaScript global function `encodeURIComponent()` to encode the value before storing it in the cookie. If you do this, you'll have to use the corresponding `decodeURIComponent()` function when you read the cookie value.

A cookie written with a simple name/value pair lasts for the current web-browsing session but is lost when the user exits the browser. To create a cookie that can last across browser sessions, specify its lifetime (in seconds) with a `max-age` attribute. You can do this by setting the `cookie` property to a string of the form: `name=value; max-age=seconds`. The following function sets a cookie with an optional `max-age` attribute:

```
// Store the name/value pair as a cookie, encoding the value with
// encodeURIComponent() in order to escape semicolons, commas, and spaces.
// If daysToLive is a number, set the max-age attribute so that the cookie
// expires after the specified number of days. Pass 0 to delete a cookie.
function setCookie(name, value, daysToLive=null) {
    let cookie = `${name}=${encodeURIComponent(value)}`;
    if (daysToLive !== null) {
        cookie += `; max-age=${daysToLive*60*60*24}`;
    }
    document.cookie = cookie;
}
```

Similarly, you can set the `path` and `domain` attributes of a cookie by appending strings of the form `;path=value` or `;domain=value` to the string that you set on the `document.cookie` property. To set the `secure` property, simply append `;secure`.

To change the value of a cookie, set its value again using the same name, path, and domain along with the new value. You can change the lifetime of a cookie when you change its value by specifying a new `max-age` attribute.

To delete a cookie, set it again using the same name, path, and domain, specifying an arbitrary (or empty) value, and a `max-age` attribute of 0.

15.12.3 IndexedDB

Web application architecture has traditionally featured HTML, CSS, and JavaScript on the client and a database on the server. You may find it surprising, therefore, to learn that the web platform includes a simple object database with a JavaScript API for persistently storing JavaScript objects on the user's computer and retrieving them as needed.

IndexedDB is an object database, not a relational database, and it is much simpler than databases that support SQL queries. It is more powerful, efficient, and robust than the key/value storage provided by the `localStorage`, however. Like the `localStorage`, IndexedDB databases are scoped to the origin of the containing document: two web pages with the same origin can access each other's data, but web pages from different origins cannot.

Each origin can have any number of IndexedDB databases. Each one has a name that must be unique within the origin. In the IndexedDB API, a database is simply a collection of named *object stores*. As the name implies, an object store stores objects. Objects are serialized into the object store using the structured clone algorithm (see "The Structured Clone Algorithm" on page 513), which means that the objects you store can have properties whose values are Maps, Sets, or typed arrays. Each object must have a *key* by which it can be sorted and retrieved from the store. Keys must be unique—two objects in the same store may not have the same key—and they must have a natural ordering so that they can be sorted. JavaScript strings, numbers, and

Date objects are valid keys. An IndexedDB database can automatically generate a unique key for each object you insert into the database. Often, though, the objects you insert into an object store will already have a property that is suitable for use as a key. In this case, you specify a "key path" for that property when you create the object store. Conceptually, a key path is a value that tells the database how to extract an object's key from the object.

In addition to retrieving objects from an object store by their primary key value, you may want to be able to search based on the value of other properties in the object. In order to be able to do this, you can define any number of *indexes* on the object store. (The ability to index an object store explains the name "IndexedDB.") Each index defines a secondary key for the stored objects. These indexes are not generally unique, and multiple objects may match a single key value.

IndexedDB provides atomicity guarantees: queries and updates to the database are grouped within a *transaction* so that they all succeed together or all fail together and never leave the database in an undefined, partially updated state. Transactions in IndexedDB are simpler than in many database APIs; we'll mention them again later.

Conceptually, the IndexedDB API is quite simple. To query or update a database, you first open the database you want (specifying it by name). Next, you create a transaction object and use that object to look up the desired object store within the database, also by name. Finally, you look up an object by calling the `get()` method of the object store or store a new object by calling `put()` (or by calling `add()`, if you want to avoid overwriting existing objects).

If you want to look up the objects for a range of keys, you create an IDBRange object that specifies the upper and lower bounds of the range and pass it to the `getAll()` or `openCursor()` methods of the object store.

If you want to make a query using a secondary key, you look up the named index of the object store, then call the `get()`, `getAll()`, or `openCursor()` methods of the index object, passing either a single key or an IDBRange object.

This conceptual simplicity of the IndexedDB API is complicated, however, by the fact that the API is asynchronous (so that web apps can use it without blocking the browser's main UI thread). IndexedDB was defined before Promises were widely supported, so the API is event-based rather than Promise-based, which means that it does not work with `async` and `await`.

Creating transactions and looking up object stores and indexes are synchronous operations. But opening a database, updating an object store, and querying a store or index are all asynchronous operations. These asynchronous methods all immediately return a request object. The browser triggers a success or error event on the request object when the request succeeds or fails, and you can define handlers with the `onsuccess` and `onerror` properties. Inside an `onsuccess` handler, the result of the operation

is available as the `result` property of the request object. Another useful event is the "complete" event dispatched on transaction objects when a transaction has completed successfully.

One convenient feature of this asynchronous API is that it simplifies transaction management. The IndexedDB API forces you to create a transaction object in order to get the object store on which you can perform queries and updates. In a synchronous API, you would expect to explicitly mark the end of the transaction by calling a `commit()` method. But with IndexedDB, transactions are automatically committed (if you do not explicitly abort them) when all the `onsuccess` event handlers have run and there are no more pending asynchronous requests that refer to that transaction.

There is one more event that is important to the IndexedDB API. When you open a database for the first time, or when you increment the version number of an existing database, IndexedDB fires an "upgradeneeded" event on the request object returned by the `indexedDB.open()` call. The job of the event handler for "upgradeneeded" events is to define or update the schema for the new database (or the new version of the existing database). For IndexedDB databases, this means creating object stores and defining indexes on those object stores. And in fact, the only time the IndexedDB API allows you to create an object store or an index is in response to an "upgradeneeded" event.

With this high-level overview of IndexedDB in mind, you should now be able to understand Example 15-13. That example uses IndexedDB to create and query a database that maps US postal codes (zip codes) to US cities. It demonstrates many, but not all, of the basic features of IndexedDB. Example 15-13 is long, but well commented.

Example 15-13. A IndexedDB database of US postal codes

```
// This utility function asynchronously obtains the database object (creating
// and initializing the DB if necessary) and passes it to the callback.
function withDB(callback) {
    let request = indexedDB.open("zipcodes", 1); // Request v1 of the database
    request.onerror = console.error;   // Log any errors
    request.onsuccess = () => {   // Or call this when done
        let db = request.result;  // The result of the request is the database
        callback(db);             // Invoke the callback with the database
    };

    // If version 1 of the database does not yet exist, then this event
    // handler will be triggered. This is used to create and initialize
    // object stores and indexes when the DB is first created or to modify
    // them when we switch from one version of the DB schema to another.
    request.onupgradeneeded = () => { initdb(request.result, callback); };
}
```

```
// withDB() calls this function if the database has not been initialized yet.
// We set up the database and populate it with data, then pass the database to
// the callback function.
//
// Our zip code database includes one object store that holds objects like this:
//
//   {
//     zipcode: "02134",
//     city: "Allston",
//     state: "MA",
//   }
//
// We use the "zipcode" property as the database key and create an index for
// the city name.
function initdb(db, callback) {
    // Create the object store, specifying a name for the store and
    // an options object that includes the "key path" specifying the
    // property name of the key field for this store.
    let store = db.createObjectStore("zipcodes", // store name
                                     { keyPath: "zipcode" });

    // Now index the object store by city name as well as by zip code.
    // With this method the key path string is passed directly as a
    // required argument rather than as part of an options object.
    store.createIndex("cities", "city");

    // Now get the data we are going to initialize the database with.
    // The zipcodes.json data file was generated from CC-licensed data from
    // www.geonames.org: https://download.geonames.org/export/zip/US.zip
    fetch("zipcodes.json")                  // Make an HTTP GET request
        .then(response => response.json())  // Parse the body as JSON
        .then(zipcodes => {                 // Get 40K zip code records
            // In order to insert zip code data into the database we need a
            // transaction object. To create our transaction object, we need
            // to specify which object stores we'll be using (we only have
            // one) and we need to tell it that we'll be doing writes to the
            // database, not just reads:
            let transaction = db.transaction(["zipcodes"], "readwrite");
            transaction.onerror = console.error;

            // Get our object store from the transaction
            let store = transaction.objectStore("zipcodes");

            // The best part about the IndexedDB API is that object stores
            // are *really* simple. Here's how we add (or update) our records:
            for(let record of zipcodes) { store.put(record); }

            // When the transaction completes successfully, the database
            // is initialized and ready for use, so we can call the
            // callback function that was originally passed to withDB()
            transaction.oncomplete = () => { callback(db); };
        });
```

```
}

// Given a zip code, use the IndexedDB API to asynchronously look up the city
// with that zip code, and pass it to the specified callback, or pass null if
// no city is found.
function lookupCity(zip, callback) {
    withDB(db => {
        // Create a read-only transaction object for this query. The
        // argument is an array of object stores we will need to use.
        let transaction = db.transaction(["zipcodes"]);

        // Get the object store from the transaction
        let zipcodes = transaction.objectStore("zipcodes");

        // Now request the object that matches the specified zipcode key.
        // The lines above were synchronous, but this one is async.
        let request = zipcodes.get(zip);
        request.onerror = console.error;  // Log errors
        request.onsuccess = () => {       // Or call this function on success
            let record = request.result;  // This is the query result
            if (record) { // If we found a match, pass it to the callback
                callback(`${record.city}, ${record.state}`);
            } else {     // Otherwise, tell the callback that we failed
                callback(null);
            }
        };
    });
}

// Given the name of a city, use the IndexedDB API to asynchronously
// look up all zip code records for all cities (in any state) that have
// that (case-sensitive) name.
function lookupZipcodes(city, callback) {
    withDB(db => {
        // As above, we create a transaction and get the object store
        let transaction = db.transaction(["zipcodes"]);
        let store = transaction.objectStore("zipcodes");

        // This time we also get the city index of the object store
        let index = store.index("cities");

        // Ask for all matching records in the index with the specified
        // city name, and when we get them we pass them to the callback.
        // If we expected more results, we might use openCursor() instead.
        let request = index.getAll(city);
        request.onerror = console.error;
        request.onsuccess = () => { callback(request.result); };
    });
}
```

15.13 Worker Threads and Messaging

One of the fundamental features of JavaScript is that it is single-threaded: a browser will never run two event handlers at the same time, and it will never trigger a timer while an event handler is running, for example. Concurrent updates to application state or to the document are simply not possible, and client-side programmers do not need to think about, or even understand, concurrent programming. A corollary is that client-side JavaScript functions must not run too long; otherwise, they will tie up the event loop and the web browser will become unresponsive to user input. This is the reason that `fetch()` is an asynchronous function, for example.

Web browsers very carefully relax the single-thread requirement with the Worker class: instances of this class represent threads that run concurrently with the main thread and the event loop. Workers live in a self-contained execution environment with a completely independent global object and no access to the Window or Document objects. Workers can communicate with the main thread only through asynchronous message passing. This means that concurrent modifications of the DOM remain impossible, but it also means that you can write long-running functions that do not stall the event loop and hang the browser. Creating a new worker is not a heavyweight operation like opening a new browser window, but workers are not flyweight "fibers" either, and it does not make sense to create new workers to perform trivial operations. Complex web applications may find it useful to create tens of workers, but it is unlikely that an application with hundreds or thousands of workers would be practical.

Workers are useful when your application needs to perform computationally intensive tasks, such as image processing. Using a worker moves tasks like this off the main thread so that the browser does not become unresponsive. And workers also offer the possibility of dividing the work among multiple threads. But workers are also useful when you have to perform frequent moderately intensive computations. Suppose, for example, that you're implementing a simple in-browser code editor, and want to include syntax highlighting. To get the highlighting right, you need to parse the code on every keystroke. But if you do that on the main thread, it is likely that the parsing code will prevent the event handlers that respond to the user's key strokes from running promptly and the user's typing experience will be sluggish.

As with any threading API, there are two parts to the Worker API. The first is the Worker object: this is what a worker looks like from the outside, to the thread that creates it. The second is the WorkerGlobalScope: this is the global object for a new worker, and it is what a worker thread looks like, on the inside, to itself.

The following sections cover Worker and WorkerGlobalScope and also explain the message-passing API that allows workers to communicate with the main thread and each other. The same communication API is used to exchange messages between a

document and <iframe> elements contained in the document, and this is covered in the following sections as well.

15.13.1 Worker Objects

To create a new worker, call the Worker() constructor, passing a URL that specifies the JavaScript code that the worker is to run:

```
let dataCruncher = new Worker("utils/cruncher.js");
```

If you specify a relative URL, it is resolved relative to the URL of the document that contains the script that called the Worker() constructor. If you specify an absolute URL, it must have the same origin (same protocol, host, and port) as that containing document.

Once you have a Worker object, you can send data to it with postMessage(). The value you pass to postMessage() will be copied using the structured clone algorithm (see "The Structured Clone Algorithm" on page 513), and the resulting copy will be delivered to the worker via a message event:

```
dataCruncher.postMessage("/api/data/to/crunch");
```

Here we're just passing a single string message, but you can also use objects, arrays, typed arrays, Maps, Sets, and so on. You can receive messages from a worker by listening for "message" events on the Worker object:

```
dataCruncher.onmessage = function(e) {
    let stats = e.data;  // The message is the data property of the event
    console.log(`Average: ${stats.mean}`);
}
```

Like all event targets, Worker objects define the standard addEventListener() and removeEventListener() methods, and you can use these in place of the onmessage.

In addition to postMessage(), Worker objects have just one other method, terminate(), which forces a worker thread to stop running.

15.13.2 The Global Object in Workers

When you create a new worker with the Worker() constructor, you specify the URL of a file of JavaScript code. That code is executed in a new, pristine JavaScript execution environment, isolated from the script that created the worker. The global object for that new execution environment is a WorkerGlobalScope object. A WorkerGlobalScope is something more than the core JavaScript global object, but less than a full-blown client-side Window object.

The WorkerGlobalScope object has a postMessage() method and an onmessage event handler property that are just like those of the Worker object but work in the

opposite direction: calling `postMessage()` inside a worker generates a message event outside the worker, and messages sent from outside the worker are turned into events and delivered to the `onmessage` handler. Because the WorkerGlobalScope is the global object for a worker, `postMessage()` and `onmessage` look like a global function and global variable to worker code.

If you pass an object as the second argument to the `Worker()` constructor, and if that object has a `name` property, then the value of that property becomes the value of the `name` property in the worker's global object. A worker might include this name in any messages it prints with `console.warn()` or `console.error()`.

The `close()` function allows a worker to terminate itself, and it is similar in effect to the `terminate()` method of a Worker object.

Since WorkerGlobalScope is the global object for workers, it has all of the properties of the core JavaScript global object, such as the JSON object, the `isNaN()` function, and the `Date()` constructor. In addition, however, WorkerGlobalScope also has the following properties of the client-side Window object:

- `self` is a reference to the global object itself. WorkerGlobalScope is not a Window object and does not define a `window` property.
- The timer methods `setTimeout()`, `clearTimeout()`, `setInterval()`, and `clearInterval()`.
- A `location` property that describes the URL that was passed to the `Worker()` constructor. This property refers to a Location object, just as the `location` property of a Window does. The Location object has properties `href`, `protocol`, `host`, `hostname`, `port`, `pathname`, `search`, and `hash`. In a worker, these properties are read-only, however.
- A `navigator` property that refers to an object with properties like those of the Navigator object of a window. A worker's Navigator object has the properties `appName`, `appVersion`, `platform`, `userAgent`, and `onLine`.
- The usual event target methods `addEventListener()` and `removeEventListener()`.

Finally, the WorkerGlobalScope object includes important client-side JavaScript APIs including the Console object, the `fetch()` function, and the IndexedDB API. WorkerGlobalScope also includes the `Worker()` constructor, which means that worker threads can create their own workers.

15.13.3 Importing Code into a Worker

Workers were defined in web browsers before JavaScript had a module system, so workers have a unique system for including additional code. WorkerGlobalScope defines `importScripts()` as a global function that all workers have access to:

```
// Before we start working, load the classes and utilities we'll need
importScripts("utils/Histogram.js", "utils/BitSet.js");
```

`importScripts()` takes one or more URL arguments, each of which should refer to a file of JavaScript code. Relative URLs are resolved relative to the URL that was passed to the `Worker()` constructor (not relative to the containing document). `importScripts()` synchronously loads and executes these files one after the other, in the order in which they were specified. If loading a script causes a network error, or if executing throws an error of any sort, none of the subsequent scripts are loaded or executed. A script loaded with `importScripts()` can itself call `importScripts()` to load the files it depends on. Note, however, that `importScripts()` does not try to keep track of what scripts have already loaded and does nothing to prevent dependency cycles.

`importScripts()` is a synchronous function: it does not return until all of the scripts have loaded and executed. You can start using the scripts you loaded as soon as `importScripts()` returns: there is no need for a callback, event handler, `then()` method or `await`. Once you have internalized the asynchronous nature of client-side JavaScript, it feels strange to go back to simple, synchronous programming again. But that is the beauty of threads: you can use a blocking function call in a worker without blocking the event loop in the main thread, and without blocking the computations being concurrently performed in other workers.

Modules in Workers

In order to use modules in workers, you must pass a second argument to the `Worker()` constructor. This second argument must be an object with a `type` property set to the string "module." Passing a `type:"module"` option to the `Worker()` constructor is much like using the `type="module"` attribute on an HTML `<script>` tag: it means that the code should be interpreted as a module and that `import` declarations are allowed.

When a worker loads a module instead of a traditional script, the WorkerGlobalScope does not define the `importScripts()` function.

Note that as of early 2020, Chrome is the only browser that supports true modules and `import` declarations in workers.

15.13.4 Worker Execution Model

Worker threads run their code (and all imported scripts or modules) synchronously from top to bottom, and then enter an asynchronous phase in which they respond to events and timers. If a worker registers a "message" event handler, it will never exit as long as there is a possibility that message events will still arrive. But if a worker doesn't listen for messages, it will run until there are no further pending tasks (such as `fetch()` promises and timers) and all task-related callbacks have been called. Once all registered callbacks have been called, there is no way a worker can begin a new task, so it is safe for the thread to exit, which it will do automatically. A worker can also explicitly shut itself down by calling the global `close()` function. Note that there are no properties or methods on the Worker object that specify whether a worker thread is still running or not, so workers should not close themselves without somehow coordinating this with their parent thread.

Errors in Workers

If an exception occurs in a worker and is not caught by any `catch` clause, then an "error" event is triggered on the global object of the worker. If this event is handled and the handler calls the `preventDefault()` method of the event object, the error propagation ends. Otherwise, the "error" event is fired on the Worker object. If `preventDefault()` is called there, then propagation ends. Otherwise, an error message is printed in the developer console and the onerror handler (§15.1.7) of the Window object is invoked.

```
// Handle uncaught worker errors with a handler inside the worker.
self.onerror = function(e) {
    console.log(`Error in worker at ${e.filename}:${e.lineno}: ${e.message}`);
    e.preventDefault();
};

// Or, handle uncaught worker errors with a handler outside the worker.
worker.onerror = function(e) {
    console.log(`Error in worker at ${e.filename}:${e.lineno}: ${e.message}`);
    e.preventDefault();
};
```

Like windows, workers can register a handler to be invoked when a Promise is rejected and there is no `.catch()` function to handle it. Within a worker you can detect this by defining a `self.onunhandledrejection` function or by using `addEventListener()` to register a global handler for "unhandledrejection" events. The event object passed to this handler will have a `promise` property whose value is the Promise object that rejected and a `reason` property whose value is what would have been passed to a `.catch()` function.

15.13.5 postMessage(), MessagePorts, and MessageChannels

The `postMessage()` method of the Worker object and the global `postMesage()` function defined inside a worker both work by invoking the `postMessage()` methods of a pair of MessagePort objects that are automatically created along with the worker. Client-side JavaScript can't directly access these automatically created MessagePort objects, but it can create new pairs of connected ports with the `MessageChannel()` constructor:

```
let channel = new MessageChannel;               // Create a new channel.
let myPort = channel.port1;                     // It has two ports
let yourPort = channel.port2;                   // connected to each other.

myPort.postMessage("Can you hear me?");         // A message posted to one will
yourPort.onmessage = (e) => console.log(e.data); // be received on the other.
```

A MessageChannel is an object with `port1` and `port2` properties that refer to a pair of connected MessagePort objects. A MessagePort is an object with a `postMessage()` method and an `onmessage` event handler property. When `postMessage()` is called on one port of a connected pair, a "message" event is fired on the other port in the pair. You can receive these "message" events by setting the `onmessage` property or by using `addEventListener()` to register a listener for "message" events.

Messages sent to a port are queued until the `onmessage` property is defined or until the `start()` method is called on the port. This prevents messages sent by one end of the channel from being missed by the other end. If you use `addEventListener()` with a MessagePort, don't forget to call `start()` or you may never see a message delivered.

All the `postMessage()` calls we've seen so far have taken a single message argument. But the method also accepts an optional second argument. This second argument is an array of items that are to be transferred to the other end of the channel instead of having a copy sent across the channel. Values that can be transferred instead of copied are MessagePorts and ArrayBuffers. (Some browsers also implement other transferable types, such as ImageBitmap and OffscreenCanvas. These are not universally supported, however, and are not covered in this book.) If the first argument to `postMessage()` includes a MessagePort (nested anywhere within the message object), then that MessagePort must also appear in the second argument. If you do this, then the MessagePort will become available to the other end of the channel and will immediately become nonfunctional on your end. Suppose you have created a worker and want to have two channels for communicating with it: one channel for ordinary data exchange and one channel for high-priority messages. In the main thread, you might create a MessageChannel, then call `postMessage()` on the worker to pass one of the MessagePorts to it:

```
let worker = new Worker("worker.js");
let urgentChannel = new MessageChannel();
```

```
let urgentPort = urgentChannel.port1;
worker.postMessage({ command: "setUrgentPort", value: urgentChannel.port2 },
                   [ urgentChannel.port2 ]);
// Now we can receive urgent messages from the worker like this
urgentPort.addEventListener("message", handleUrgentMessage);
urgentPort.start();  // Start receiving messages
// And send urgent messages like this
urgentPort.postMessage("test");
```

MessageChannels are also useful if you create two workers and want to allow them to communicate directly with each other rather than requiring code on the main thread to relay messages between them.

The other use of the second argument to `postMessage()` is to transfer ArrayBuffers between workers without having to copy them. This is an important performance enhancement for large ArrayBuffers like those used to hold image data. When an ArrayBuffer is transferred over a MessagePort, the ArrayBuffer becomes unusable in the original thread so that there is no possibility of concurrent access to its contents. If the first argument to `postMessage()` includes an ArrayBuffer, or any value (such as a typed array) that has an ArrayBuffer, then that buffer may appear as an array element in the second `postMessage()` argument. If it does appear, then it will be transferred without copying. If not, then the ArrayBuffer will be copied rather than transferred. Example 15-14 will demonstrate the use of this transfer technique with ArrayBuffers.

15.13.6 Cross-Origin Messaging with postMessage()

There is another use case for the `postMessage()` method in client-side JavaScript. It involves windows instead of workers, but there are enough similarities between the two cases that we will describe the `postMessage()` method of the Window object here.

When a document contains an `<iframe>` element, that element acts as an embedded but independent window. The Element object that represents the `<iframe>` has a `contentWindow` property that is the Window object for the embedded document. And for scripts running within that nested iframe, the `window.parent` property refers to the containing Window object. When two windows display documents with the same origin, then scripts in each of those windows have access to the contents of the other window. But when the documents have different origins, the browser's same-origin policy prevents JavaScript in one window from accessing the content of another window.

For workers, `postMessage()` provides a safe way for two independent threads to communicate without sharing memory. For windows, `postMessage()` provides a controlled way for two independent origins to safely exchange messages. Even if the same-origin policy prevents your script from seeing the content of another window,

you can still call `postMessage()` on that window, and doing so will cause a "message" event to be triggered on that window, where it can be seen by the event handlers in that window's scripts.

The `postMessage()` method of a Window is a little different than the `postMessage()` method of a Worker, however. The first argument is still an arbitrary message that will be copied by the structured clone algorithm. But the optional second argument listing objects to be transferred instead of copied becomes an optional third argument. The `postMessage()` method of a window takes a string as its required second argument. This second argument should be an origin (a protocol, hostname, and optional port) that specifies who you expect to be receiving the message. If you pass the string "https://good.example.com" as the second argument, but the window you are posting the message to actually contains content from "https://malware.example.com," then the message you posted will not be delivered. If you are willing to send your message to content from any origin, then you can pass the wildcard "*" as the second argument.

JavaScript code running inside a window or `<iframe>` can receive messages posted to that window or frame by defining the `onmessage` property of that window or by calling `addEventListener()` for "message" events. As with workers, when you receive a "message" event for a window, the `data` property of the event object is the message that was sent. In addition, however, "message" events delivered to windows also define `source` and `origin` properties. The `source` property specifies the Window object that sent the event, and you can use `event.source.postMessage()` to send a reply. The `origin` property specifies the origin of the content in the source window. This is not something the sender of the message can forge, and when you receive a "message" event, you will typically want to verify that it is from an origin you expect.

15.14 Example: The Mandelbrot Set

This chapter on client-side JavaScript culminates with a long example that demonstrates using workers and messaging to parallelize computationally intensive tasks. But it is written to be an engaging, real-world web application and also demonstrates a number of the other APIs demonstrated in this chapter, including history management; use of the ImageData class with a `<canvas>`; and the use of keyboard, pointer, and resize events. It also demonstrates important core JavaScript features, including generators and a sophisticated use of Promises.

The example is a program for displaying and exploring the Mandelbrot set, a complex fractal that includes beautiful images like the one shown in Figure 15-16.

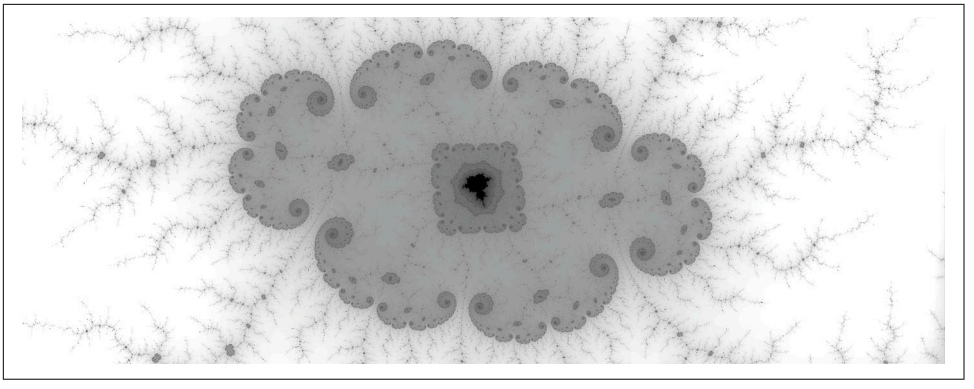

Figure 15-16. A portion of the Mandelbrot set

The Mandelbrot set is defined as the set of points on the complex plane, which, when put through a repeated process of complex multiplication and addition, produce a value whose magnitude remains bounded. The contours of the set are surprisingly complex, and computing which points are members of the set and which are not is computationally intensive: to produce a 500×500 image of the Mandelbrot set, you must individually compute the membership of each of the 250,000 pixels in your image. And to verify that the value associated with each pixel remains bounded, you may have to repeat the process of complex multiplication 1,000 times or more. (More iterations give more sharply defined boundaries for the set; fewer iterations produce fuzzier boundaries.) With up to 250 million steps of complex arithmetic required to produce a high-quality image of the Mandelbrot set, you can understand why using workers is a valuable technique. Example 15-14 shows the worker code we will use. This file is relatively compact: it is just the raw computational muscle for the larger program. Two things are worth noting about it, however:

- The worker creates an ImageData object to represent the rectangular grid of pixels for which it is computing Mandelbrot set membership. But instead of storing actual pixel values in the ImageData, it uses a custom-typed array to treat each pixel as a 32-bit integer. It stores the number of iterations required for each pixel in this array. If the magnitude of the complex number computed for each pixel becomes greater than four, then it is mathematically guaranteed to grow without bounds from then on, and we say it has "escaped." So the value this worker returns for each pixel is the number of iterations before the value escaped. We tell the worker the maximum number of iterations it should try for each value, and pixels that reach this maximum number are considered to be in the set.
- The worker transfers the ArrayBuffer associated with the ImageData back to the main thread so the memory associated with it does not need to be copied.

Example 15-14. Worker code for computing regions of the Mandelbrot set

```
// This is a simple worker that receives a message from its parent thread,
// performs the computation described by that message and then posts the
// result of that computation back to the parent thread.
onmessage = function(message) {
    // First, we unpack the message we received:
    //  - tile is an object with width and height properties. It specifies the
    //    size of the rectangle of pixels for which we will be computing
    //    Mandelbrot set membership.
    //  - (x0, y0) is the point in the complex plane that corresponds to the
    //    upper-left pixel in the tile.
    //  - perPixel is the pixel size in both the real and imaginary dimensions.
    //  - maxIterations specifies the maximum number of iterations we will
    //    perform before deciding that a pixel is in the set.
    const {tile, x0, y0, perPixel, maxIterations} = message.data;
    const {width, height} = tile;

    // Next, we create an ImageData object to represent the rectangular array
    // of pixels, get its internal ArrayBuffer, and create a typed array view
    // of that buffer so we can treat each pixel as a single integer instead of
    // four individual bytes. We'll store the number of iterations for each
    // pixel in this iterations array. (The iterations will be transformed into
    // actual pixel colors in the parent thread.)
    const imageData = new ImageData(width, height);
    const iterations = new Uint32Array(imageData.data.buffer);

    // Now we begin the computation. There are three nested for loops here.
    // The outer two loop over the rows and columns of pixels, and the inner
    // loop iterates each pixel to see if it "escapes" or not. The various
    // loop variables are the following:
    // - row and column are integers representing the pixel coordinate.
    // - x and y represent the complex point for each pixel: x + yi.
    // - index is the index in the iterations array for the current pixel.
    // - n tracks the number of iterations for each pixel.
    // - max and min track the largest and smallest number of iterations
    //   we've seen so far for any pixel in the rectangle.
    let index = 0, max = 0, min=maxIterations;
    for(let row = 0, y = y0; row < height; row++, y += perPixel) {
        for(let column = 0, x = x0; column < width; column++, x += perPixel) {
            // For each pixel we start with the complex number c = x+yi.
            // Then we repeatedly compute the complex number z(n+1) based on
            // this recursive formula:
            //    z(0) = c
            //    z(n+1) = z(n)^2 + c
            // If |z(n)| (the magnitude of z(n)) is > 2, then the
            // pixel is not part of the set and we stop after n iterations.
            let n;             // The number of iterations so far
            let r = x, i = y;  // Start with z(0) set to c
            for(n = 0; n < maxIterations; n++) {
                let rr = r*r, ii = i*i; // Square the two parts of z(n).
                if (rr + ii > 4) {      // If |z(n)|^2 is > 4 then
```

```
                    break;              // we've escaped and can stop iterating.
                }
                i = 2*r*i + y;          // Compute imaginary part of z(n+1).
                r = rr - ii + x;        // And the real part of z(n+1).
            }
            iterations[index++] = n;    // Remember # iterations for each pixel.
            if (n > max) max = n;       // Track the maximum number we've seen.
            if (n < min) min = n;       // And the minimum as well.
        }
    }

    // When the computation is complete, send the results back to the parent
    // thread. The imageData object will be copied, but the giant ArrayBuffer
    // it contains will be transferred for a nice performance boost.
    postMessage({tile, imageData, min, max}, [imageData.data.buffer]);
};
```

The Mandelbrot set viewer application that uses that worker code is shown in Example 15-15. Now that you have nearly reached the end of this chapter, this long example is something of a capstone experience that brings together a number of important core and client-side JavaScript features and APIs. The code is thoroughly commented, and I encourage you to read it carefully.

Example 15-15. A web application for displaying and exploring the Mandelbrot set

```
/*
 * This class represents a subrectangle of a canvas or image. We use Tiles to
 * divide a canvas into regions that can be processed independently by Workers.
 */
class Tile {
    constructor(x, y, width, height) {
        this.x = x;                     // The properties of a Tile object
        this.y = y;                     // represent the position and size
        this.width = width;             // of the tile within a larger
        this.height = height;           // rectangle.
    }

    // This static method is a generator that divides a rectangle of the
    // specified width and height into the specified number of rows and
    // columns and yields numRows*numCols Tile objects to cover the rectangle.
    static *tiles(width, height, numRows, numCols) {
        let columnWidth = Math.ceil(width / numCols);
        let rowHeight = Math.ceil(height / numRows);

        for(let row = 0; row < numRows; row++) {
            let tileHeight = (row < numRows-1)
                ? rowHeight                         // height of most rows
                : height - rowHeight * (numRows-1);  // height of last row
            for(let col = 0; col < numCols; col++) {
                let tileWidth = (col < numCols-1)
                    ? columnWidth                   // width of most columns
```

```
                    : width - columnWidth * (numCols-1); // and last column

                yield new Tile(col * columnWidth, row * rowHeight,
                               tileWidth, tileHeight);
            }
        }
    }
}

/*
 * This class represents a pool of workers, all running the same code. The
 * worker code you specify must respond to each message it receives by
 * performing some kind of computation and then posting a single message with
 * the result of that computation.
 *
 * Given a WorkerPool and message that represents work to be performed, simply
 * call addWork(), with the message as an argument. If there is a Worker
 * object that is currently idle, the message will be posted to that worker
 * immediately. If there are no idle Worker objects, the message will be
 * queued and will be posted to a Worker when one becomes available.
 *
 * addWork() returns a Promise, which will resolve with the message recieved
 * from the work, or will reject if the worker throws an unhandled error.
 */
class WorkerPool {
    constructor(numWorkers, workerSource) {
        this.idleWorkers = [];       // Workers that are not currently working
        this.workQueue = [];         // Work not currently being processed
        this.workerMap = new Map();  // Map workers to resolve and reject funcs

        // Create the specified number of workers, add message and error
        // handlers and save them in the idleWorkers array.
        for(let i = 0; i < numWorkers; i++) {
            let worker = new Worker(workerSource);
            worker.onmessage = message => {
                this._workerDone(worker, null, message.data);
            };
            worker.onerror = error => {
                this._workerDone(worker, error, null);
            };
            this.idleWorkers[i] = worker;
        }
    }

    // This internal method is called when a worker finishes working, either
    // by sending a message or by throwing an error.
    _workerDone(worker, error, response) {
        // Look up the resolve() and reject() functions for this worker
        // and then remove the worker's entry from the map.
        let [resolver, rejector] = this.workerMap.get(worker);
        this.workerMap.delete(worker);
```

```
            // If there is no queued work, put this worker back in
            // the list of idle workers. Otherwise, take work from the queue
            // and send it to this worker.
            if (this.workQueue.length === 0) {
                this.idleWorkers.push(worker);
            } else {
                let [work, resolver, rejector] = this.workQueue.shift();
                this.workerMap.set(worker, [resolver, rejector]);
                worker.postMessage(work);
            }

            // Finally, resolve or reject the promise associated with the worker.
            error === null ? resolver(response) : rejector(error);
        }

        // This method adds work to the worker pool and returns a Promise that
        // will resolve with a worker's response when the work is done. The work
        // is a value to be passed to a worker with postMessage(). If there is an
        // idle worker, the work message will be sent immediately. Otherwise it
        // will be queued until a worker is available.
        addWork(work) {
            return new Promise((resolve, reject) => {
                if (this.idleWorkers.length > 0) {
                    let worker = this.idleWorkers.pop();
                    this.workerMap.set(worker, [resolve, reject]);
                    worker.postMessage(work);
                } else {
                    this.workQueue.push([work, resolve, reject]);
                }
            });
        }
    }

    /*
     * This class holds the state information necessary to render a Mandelbrot set.
     * The cx and cy properties give the point in the complex plane that is the
     * center of the image. The perPixel property specifies how much the real and
     * imaginary parts of that complex number changes for each pixel of the image.
     * The maxIterations property specifies how hard we work to compute the set.
     * Larger numbers require more computation but produce crisper images.
     * Note that the size of the canvas is not part of the state. Given cx, cy, and
     * perPixel we simply render whatever portion of the Mandelbrot set fits in
     * the canvas at its current size.
     *
     * Objects of this type are used with history.pushState() and are used to read
     * the desired state from a bookmarked or shared URL.
     */
    class PageState {
        // This factory method returns an initial state to display the entire set.
        static initialState() {
            let s = new PageState();
            s.cx = -0.5;
```

```
        s.cy = 0;
        s.perPixel = 3/window.innerHeight;
        s.maxIterations = 500;
        return s;
    }

    // This factory method obtains state from a URL, or returns null if
    // a valid state could not be read from the URL.
    static fromURL(url) {
        let s = new PageState();
        let u = new URL(url); // Initialize state from the url's search params.
        s.cx = parseFloat(u.searchParams.get("cx"));
        s.cy = parseFloat(u.searchParams.get("cy"));
        s.perPixel = parseFloat(u.searchParams.get("pp"));
        s.maxIterations = parseInt(u.searchParams.get("it"));
        // If we got valid values, return the PageState object, otherwise null.
        return (isNaN(s.cx) || isNaN(s.cy) || isNaN(s.perPixel)
                || isNaN(s.maxIterations))
            ? null
            : s;
    }

    // This instance method encodes the current state into the search
    // parameters of the browser's current location.
    toURL() {
        let u = new URL(window.location);
        u.searchParams.set("cx", this.cx);
        u.searchParams.set("cy", this.cy);
        u.searchParams.set("pp", this.perPixel);
        u.searchParams.set("it", this.maxIterations);
        return u.href;
    }
}

// These constants control the parallelism of the Mandelbrot set computation.
// You may need to adjust them to get optimum performance on your computer.
const ROWS = 3, COLS = 4, NUMWORKERS = navigator.hardwareConcurrency || 2;

// This is the main class of our Mandelbrot set program. Simply invoke the
// constructor function with the <canvas> element to render into. The program
// assumes that this <canvas> element is styled so that it is always as big
// as the browser window.
class MandelbrotCanvas {
    constructor(canvas) {
        // Store the canvas, get its context object, and initialize a WorkerPool
        this.canvas = canvas;
        this.context = canvas.getContext("2d");
        this.workerPool = new WorkerPool(NUMWORKERS, "mandelbrotWorker.js");

        // Define some properties that we'll use later
        this.tiles = null;          // Subregions of the canvas
        this.pendingRender = null;  // We're not currently rendering
```

```
        this.wantsRerender = false; // No render is currently requested
        this.resizeTimer = null;    // Prevents us from resizing too frequently
        this.colorTable = null;     // For converting raw data to pixel values.

        // Set up our event handlers
        this.canvas.addEventListener("pointerdown", e => this.handlePointer(e));
        window.addEventListener("keydown", e => this.handleKey(e));
        window.addEventListener("resize", e => this.handleResize(e));
        window.addEventListener("popstate", e => this.setState(e.state, false));

        // Initialize our state from the URL or start with the initial state.
        this.state =
            PageState.fromURL(window.location) || PageState.initialState();

        // Save this state with the history mechanism.
        history.replaceState(this.state, "", this.state.toURL());

        // Set the canvas size and get an array of tiles that cover it.
        this.setSize();

        // And render the Mandelbrot set into the canvas.
        this.render();
    }

    // Set the canvas size and initialize an array of Tile objects. This
    // method is called from the constructor and also by the handleResize()
    // method when the browser window is resized.
    setSize() {
        this.width = this.canvas.width = window.innerWidth;
        this.height = this.canvas.height = window.innerHeight;
        this.tiles = [...Tile.tiles(this.width, this.height, ROWS, COLS)];
    }

    // This function makes a change to the PageState, then re-renders the
    // Mandelbrot set using that new state, and also saves the new state with
    // history.pushState(). If the first argument is a function that function
    // will be called with the state object as its argument and should make
    // changes to the state. If the first argument is an object, then we simply
    // copy the properties of that object into the state object. If the optional
    // second argument is false, then the new state will not be saved. (We
    // do this when calling setState in response to a popstate event.)
    setState(f, save=true) {
        // If the argument is a function, call it to update the state.
        // Otherwise, copy its properties into the current state.
        if (typeof f === "function") {
            f(this.state);
        } else {
            for(let property in f) {
                this.state[property] = f[property];
            }
        }
```

```
        // In either case, start rendering the new state ASAP.
        this.render();

        // Normally we save the new state. Except when we're called with
        // a second argument of false which we do when we get a popstate event.
        if (save) {
            history.pushState(this.state, "", this.state.toURL());
        }
    }

    // This method asynchronously draws the portion of the Mandelbrot set
    // specified by the PageState object into the canvas. It is called by
    // the constructor, by setState() when the state changes, and by the
    // resize event handler when the size of the canvas changes.
    render() {
        // Sometimes the user may use the keyboard or mouse to request renders
        // more quickly than we can perform them. We don't want to submit all
        // the renders to the worker pool. Instead if we're rendering, we'll
        // just make a note that a new render is needed, and when the current
        // render completes, we'll render the current state, possibly skipping
        // multiple intermediate states.
        if (this.pendingRender) {        // If we're already rendering,
            this.wantsRerender = true;   // make a note to rerender later
            return;                      // and don't do anything more now.
        }

        // Get our state variables and compute the complex number for the
        // upper left corner of the canvas.
        let {cx, cy, perPixel, maxIterations} = this.state;
        let x0 = cx - perPixel * this.width/2;
        let y0 = cy - perPixel * this.height/2;

        // For each of our ROWS*COLS tiles, call addWork() with a message
        // for the code in mandelbrotWorker.js. Collect the resulting Promise
        // objects into an array.
        let promises = this.tiles.map(tile => this.workerPool.addWork({
            tile: tile,
            x0: x0 + tile.x * perPixel,
            y0: y0 + tile.y * perPixel,
            perPixel: perPixel,
            maxIterations: maxIterations
        }));

        // Use Promise.all() to get an array of responses from the array of
        // promises. Each response is the computation for one of our tiles.
        // Recall from mandelbrotWorker.js that each response includes the
        // Tile object, an ImageData object that includes iteration counts
        // instead of pixel values, and the minimum and maximum iterations
        // for that tile.
        this.pendingRender = Promise.all(promises).then(responses => {

            // First, find the overall max and min iterations over all tiles.
```

```
// We need these numbers so we can assign colors to the pixels.
let min = maxIterations, max = 0;
for(let r of responses) {
    if (r.min < min) min = r.min;
    if (r.max > max) max = r.max;
}

// Now we need a way to convert the raw iteration counts from the
// workers into pixel colors that will be displayed in the canvas.
// We know that all the pixels have between min and max iterations
// so we precompute the colors for each iteration count and store
// them in the colorTable array.

// If we haven't allocated a color table yet, or if it is no longer
// the right size, then allocate a new one.
if (!this.colorTable || this.colorTable.length !== maxIterations+1){
    this.colorTable = new Uint32Array(maxIterations+1);
}

// Given the max and the min, compute appropriate values in the
// color table. Pixels in the set will be colored fully opaque
// black. Pixels outside the set will be translucent black with higher
// iteration counts resulting in higher opacity. Pixels with
// minimum iteration counts will be transparent and the white
// background will show through, resulting in a grayscale image.
if (min === max) {                // If all the pixels are the same,
    if (min === maxIterations) {  // Then make them all black
        this.colorTable[min] = 0xFF000000;
    } else {                      // Or all transparent.
        this.colorTable[min] = 0;
    }
} else {
    // In the normal case where min and max are different, use a
    // logarithic scale to assign each possible iteration count an
    // opacity between 0 and 255, and then use the shift left
    // operator to turn that into a pixel value.
    let maxlog = Math.log(1+max-min);
    for(let i = min; i <= max; i++) {
        this.colorTable[i] =
            (Math.ceil(Math.log(1+i-min)/maxlog * 255) << 24);
    }
}

// Now translate the iteration numbers in each response's
// ImageData to colors from the colorTable.
for(let r of responses) {
    let iterations = new Uint32Array(r.imageData.data.buffer);
    for(let i = 0; i < iterations.length; i++) {
        iterations[i] = this.colorTable[iterations[i]];
    }
}
```

```
            // Finally, render all the imageData objects into their
            // corresponding tiles of the canvas using putImageData().
            // (First, though, remove any CSS transforms on the canvas that may
            // have been set by the pointerdown event handler.)
            this.canvas.style.transform = "";
            for(let r of responses) {
                this.context.putImageData(r.imageData, r.tile.x, r.tile.y);
            }
        })
        .catch((reason) => {
            // If anything went wrong in any of our Promises, we'll log
            // an error here. This shouldn't happen, but this will help with
            // debugging if it does.
            console.error("Promise rejected in render():", reason);
        })
        .finally(() => {
            // When we are done rendering, clear the pendingRender flags
            this.pendingRender = null;
            // And if render requests came in while we were busy, rerender now.
            if (this.wantsRerender) {
                this.wantsRerender = false;
                this.render();
            }
        });
    }

// If the user resizes the window, this function will be called repeatedly.
// Resizing a canvas and rerendering the Mandlebrot set is an expensive
// operation that we can't do multiple times a second, so we use a timer
// to defer handling the resize until 200ms have elapsed since the last
// resize event was received.
handleResize(event) {
    // If we were already deferring a resize, clear it.
    if (this.resizeTimer) clearTimeout(this.resizeTimer);
    // And defer this resize instead.
    this.resizeTimer = setTimeout(() => {
        this.resizeTimer = null;  // Note that resize has been handled
        this.setSize();           // Resize canvas and tiles
        this.render();            // Rerender at the new size
    }, 200);
}

// If the user presses a key, this event handler will be called.
// We call setState() in response to various keys, and setState() renders
// the new state, updates the URL, and saves the state in browser history.
handleKey(event) {
    switch(event.key) {
    case "Escape":     // Type Escape to go back to the initial state
        this.setState(PageState.initialState());
        break;
    case "+":          // Type + to increase the number of iterations
        this.setState(s => {
```

```
            s.maxIterations = Math.round(s.maxIterations*1.5);
        });
        break;
    case "-":          // Type - to decrease the number of iterations
        this.setState(s => {
            s.maxIterations = Math.round(s.maxIterations/1.5);
            if (s.maxIterations < 1) s.maxIterations = 1;
        });
        break;
    case "o":          // Type o to zoom out
        this.setState(s => s.perPixel *= 2);
        break;
    case "ArrowUp":    // Up arrow to scroll up
        this.setState(s => s.cy -= this.height/10 * s.perPixel);
        break;
    case "ArrowDown":  // Down arrow to scroll down
        this.setState(s => s.cy += this.height/10 * s.perPixel);
        break;
    case "ArrowLeft":  // Left arrow to scroll left
        this.setState(s => s.cx -= this.width/10 * s.perPixel);
        break;
    case "ArrowRight": // Right arrow to scroll right
        this.setState(s => s.cx += this.width/10 * s.perPixel);
        break;
    }
}

// This method is called when we get a pointerdown event on the canvas.
// The pointerdown event might be the start of a zoom gesture (a click or
// tap) or a pan gesture (a drag). This handler registers handlers for
// the pointermove and pointerup events in order to respond to the rest
// of the gesture. (These two extra handlers are removed when the gesture
// ends with a pointerup.)
handlePointer(event) {
    // The pixel coordinates and time of the initial pointer down.
    // Because the canvas is as big as the window, these event coordinates
    // are also canvas coordinates.
    const x0 = event.clientX, y0 = event.clientY, t0 = Date.now();

    // This is the handler for move events.
    const pointerMoveHandler = event => {
        // How much have we moved, and how much time has passed?
        let dx=event.clientX-x0, dy=event.clientY-y0, dt=Date.now()-t0;

        // If the pointer has moved enough or enough time has passed that
        // this is not a regular click, then use CSS to pan the display.
        // (We will rerender it for real when we get the pointerup event.)
        if (dx > 10 || dy > 10 || dt > 500) {
            this.canvas.style.transform = `translate(${dx}px, ${dy}px)`;
        }
    };
```

```
        // This is the handler for pointerup events
        const pointerUpHandler = event => {
            // When the pointer goes up, the gesture is over, so remove
            // the move and up handlers until the next gesture.
            this.canvas.removeEventListener("pointermove", pointerMoveHandler);
            this.canvas.removeEventListener("pointerup", pointerUpHandler);

            // How much did the pointer move, and how much time passed?
            const dx = event.clientX-x0, dy=event.clientY-y0, dt=Date.now()-t0;
            // Unpack the state object into individual constants.
            const {cx, cy, perPixel} = this.state;

            // If the pointer moved far enough or if enough time passed, then
            // this was a pan gesture, and we need to change state to change
            // the center point. Otherwise, the user clicked or tapped on a
            // point and we need to center and zoom in on that point.
            if (dx > 10 || dy > 10 || dt > 500) {
                // The user panned the image by (dx, dy) pixels.
                // Convert those values to offsets in the complex plane.
                this.setState({cx: cx - dx*perPixel, cy: cy - dy*perPixel});
            } else {
                // The user clicked. Compute how many pixels the center moves.
                let cdx = x0 - this.width/2;
                let cdy = y0 - this.height/2;

                // Use CSS to quickly and temporarily zoom in
                this.canvas.style.transform =
                    `translate(${-cdx*2}px, ${-cdy*2}px) scale(2)`;

                // Set the complex coordinates of the new center point and
                // zoom in by a factor of 2.
                this.setState(s => {
                    s.cx += cdx * s.perPixel;
                    s.cy += cdy * s.perPixel;
                    s.perPixel /= 2;
                });
            }
        };

        // When the user begins a gesture we register handlers for the
        // pointermove and pointerup events that follow.
        this.canvas.addEventListener("pointermove", pointerMoveHandler);
        this.canvas.addEventListener("pointerup", pointerUpHandler);
    }
}

// Finally, here's how we set up the canvas. Note that this JavaScript file
// is self-sufficient. The HTML file only needs to include this one <script>.
let canvas = document.createElement("canvas"); // Create a canvas element
document.body.append(canvas);                  // Insert it into the body
document.body.style = "margin:0";              // No margin for the <body>
canvas.style.width = "100%";                   // Make canvas as wide as body
```

```
canvas.style.height = "100%";                    // and as high as the body.
new MandelbrotCanvas(canvas);                    // And start rendering into it!
```

15.15 Summary and Suggestions for Further Reading

This long chapter has covered the fundamentals of client-side JavaScript programming:

- How scripts and JavaScript modules are included in web pages and how and when they are executed.
- Client-side JavaScript's asynchronous, event-driven programming model.
- The Document Object Model (DOM) that allows JavaScript code to inspect and modify the HTML content of the document it is embedded within. This DOM API is the heart of all client-side JavaScript programming.
- How JavaScript code can manipulate the CSS styles that are applied to content within the document.
- How JavaScript code can obtain the coordinates of document elements in the browser window and within the document itself.
- How to create reusable UI "Web Components" with JavaScript, HTML, and CSS using the Custom Elements and Shadow DOM APIs.
- How to display and dynamically generate graphics with SVG and the HTML `<canvas>` element.
- How to add scripted sound effects (both recorded and synthesized) to your web pages.
- How JavaScript can make the browser load new pages, go backward and forward in the user's browsing history, and even add new entries to the browsing history.
- How JavaScript programs can exchange data with web servers using the HTTP and WebSocket protocols.
- How JavaScript programs can store data in the user's browser.
- How JavaScript programs can use worker threads to achieve a safe form of concurrency.

This has been the longest chapter of the book, by far. But it cannot come close to covering all the APIs available to web browsers. The web platform is sprawling and ever-evolving, and my goal for this chapter was to introduce the most important core APIs. With the knowledge you have from this book, you are well equipped to learn and use new APIs as you need them. But you can't learn about a new API if you don't know that it exists, so the short sections that follow end the chapter with a quick list of web platform features that you might want to investigate in the future.

15.15.1 HTML and CSS

The web is built upon three key technologies: HTML, CSS, and JavaScript, and knowledge of JavaScript can take you only so far as a web developer unless you also develop your expertise with HTML and CSS. It is important to know how to use JavaScript to manipulate HTML elements and CSS styles, but that knowledge is is much more useful if you also know which HTML elements and which CSS styles to use.

So before you start exploring more JavaScript APIs, I would encourage you to invest some time in mastering the other tools in a web developer's toolkit. HTML form and input elements, for example, have sophisticated behavior that is important to understand, and the flexbox and grid layout modes in CSS are incredibly powerful.

Two topics worth paying particular attention to in this area are accessibility (including ARIA attributes) and internationalization (including support for right-to-left writing directions).

15.15.2 Performance

Once you have written a web application and released it to the world, the never-ending quest to make it fast begins. It is hard to optimize things that you can't measure, however, so it is worth familiarizing yourself with the Performance APIs. The `performance` property of the window object is the main entry point to this API. It includes a high-resolution time source `performance.now()`, and methods `performance.mark()` and `performance.measure()` for marking critical points in your code and measuring the elapsed time between them. Calling these methods creates PerformanceEntry objects that you can access with `performance.getEntries()`. Browsers add their own PerformanceEntry objects any time the browser loads a new page or fetches a file over the network, and these automatically created PerformanceEntry objects include granular timing details of your application's network performance. The related PerformanceObserver class allows you to specify a function to be invoked when new PerformanceEntry objects are created.

15.15.3 Security

This chapter introduced the general idea of how to defend against cross-site scripting (XSS) security vulnerabilities in your websites, but we did not go into much detail. The topic of web security is an important one, and you may want to spend some time learning more about it. In addition to XSS, it is worth learning about the `Content-Security-Policy` HTTP header and understanding how CSP allows you to ask the web browser to restrict the capabilities it grants to JavaScript code. Understanding CORS (Cross-Origin Resource Sharing) is also important.

15.15.4 WebAssembly

WebAssembly (or "wasm") is a low-level virtual machine bytecode format that is designed to integrate well with JavaScript interpreters in web browsers. There are compilers that allow you to compile C, C++, and Rust programs to WebAssembly bytecode and to run those programs in web browsers at close to native speed, without breaking the browser sandbox or security model. WebAssembly can export functions that can be called by JavaScript programs. A typical use case for WebAssembly would be to compile the standard C-language zlib compression library so that JavaScript code has access to high-speed compression and decompression algorithms. Learn more at *https://webassembly.org*.

15.15.5 More Document and Window Features

The Window and Document objects have a number of features that were not covered in this chapter:

- The Window object defines `alert()`, `confirm()`, and `prompt()` methods that display simple modal dialogues to the user. These methods block the main thread. The `confirm()` method synchronously returns a boolean value, and `prompt()` synchronously returns a string of user input. These are not suitable for production use but can be useful for simple projects and prototypes.
- The `navigator` and `screen` properties of the Window object were mentioned in passing at the start of this chapter, but the Navigator and Screen objects that they reference have some features that were not described here that you may find useful.
- The `requestFullscreen()` method of any Element object requests that that element (a `<video>` or `<canvas>` element, for example) be displayed in fullscreen mode. The `exitFullscreen()` method of the Document returns to normal display mode.
- The `requestAnimationFrame()` method of the Window object takes a function as its argument and will execute that function when the browser is preparing to render the next frame. When you are making visual changes (especially repeated or animated ones), wrapping your code within a call to `requestAnimationFrame()` can help to ensure that the changes are rendered smoothly and in a way that is optimized by the browser.
- If the user selects text within your document, you can obtain details of that selection with the Window method `getSelection()` and get the selected text with `getSelection().toString()`. In some browsers, `navigator.clipboard` is an object with an async API for reading and setting the content of the system

clipboard to enable copy-and-paste interactions with applications outside of the browser.

- A little-known feature of web browsers is that HTML elements with a `contenteditable="true"` attribute allow their content to be edited. The `document.execCommand()` method enables rich-text editing features for editable content.
- A MutationObserver allows JavaScript to monitor changes to, or beneath, a specified element in the document. Create a MutationObserver with the `MutationObserver()` constructor, passing the callback function that should be called when changes are made. Then call the `observe()` method of the MutationObserver to specify which parts of which element are to be monitored.
- An IntersectionObserver allows JavaScript to determine which document elements are on the screen and which are close to being on the screen. It is particularly useful for applications that want to dynamically load content on demand as the user scrolls.

15.15.6 Events

The sheer number and diversity of events supported by the web platform can be daunting. This chapter has discussed a variety of event types, but here are some more that you may find useful:

- Browsers fire "online" and "offline" events at the Window object when the browser gains or loses an internet connection.
- Browsers fire a "visiblitychange" event at the Document object when a document becomes visible or invisible (usually because a user has switched tabs). JavaScript can check `document.visibilityState` to determine whether its document is currently "visible" or "hidden."
- Browsers support a complicated API to support drag-and-drop UIs and to support data exchange with applications outside the browser. This API involves a number of events, including "dragstart," "dragover," "dragend," and "drop." This API is tricky to use correctly but useful when you need it. It is an important API to know about if you want to enable users to drag files from their desktop into your web application.
- The Pointer Lock API enables JavaScript to hide the mouse pointer and get raw mouse events as relative movement amounts rather than absolute positions on the screen. This is typically useful for games. Call `requestPointerLock()` on the element you want all mouse events directed to. After you do this, "mousemove" events delivered to that element will have `movementX` and `movementY` properties.

- The Gamepad API adds support for game controllers. Use `navigator.getGamepads()` to get connected Gamepad objects, and listen for "gamepadconnected" events on the Window object to be notified when a new controller is plugged in. The Gamepad object defines an API for querying the current state of the buttons on the controller.

15.15.7 Progressive Web Apps and Service Workers

The term *Progressive Web Apps*, or PWAs, is a buzzword that describes web applications that are built using a few key technologies. Careful documentation of these key technologies would require a book of its own, and I have not covered them in this chapter, but you should be aware of all of these APIs. It is worth noting that powerful modern APIs like these are typically designed to work only on secure HTTPS connections. Websites that are still using `http://` URLs will not be able to take advantage of these:

- A ServiceWorker is a kind of worker thread with the ability to intercept, inspect, and respond to network requests from the web application that it "services." When a web application registers a service worker, that worker's code becomes persistent in the browser's local storage, and when the user visits the associated website again, the service worker is reactivated. Service workers can cache network responses (including files of JavaScript code), which means that web applications that use service workers can effectively install themselves onto the user's computer for rapid startup and offline use. The *Service Worker Cookbook* at *https://serviceworke.rs* is a valuable resource for learning about service workers and their related technologies.
- The Cache API is designed for use by service workers (but is also available to regular JavaScript code outside of workers). It works with the Request and Response objects defined by the `fetch()` API and implements a cache of Request/Response pairs. The Cache API enables a service worker to cache the scripts and other assets of the web app it serves and can also help to enable offline use of the web app (which is particularly important for mobile devices).
- A Web Manifest is a JSON-formatted file that describes a web application including a name, a URL, and links to icons in various sizes. If your web app uses a service worker and includes a `<link rel="manifest">` tag that references a `.webmanifest` file, then browsers (particularly browsers on mobile devices) may give you the option to add an icon for the web app to your desktop or home screen.
- The Notifications API allows web apps to display notifications using the native OS notification system on both mobile and desktop devices. Notifications can include an image and text, and your code can receive an event if the user clicks

on the notification. Using this API is complicated by the fact that you must first request the user's permission to display notifications.

- The Push API allows web applications that have a service worker (and that have the user's permission) to subscribe to notifications from a server, and to display those notifications even when the application itself is not running. Push notifications are common on mobile devices, and the Push API brings web apps closer to feature parity with native apps on mobile.

15.15.8 Mobile Device APIs

There are a number of web APIs that are primarily useful for web apps running on mobile devices. (Unfortunately, a number of these APIs only work on Android devices and not iOS devices.)

- The Geolocation API allows JavaScript (with the user's permission) to determine the user's physical location. It is well supported on desktop and mobile devices, including iOS devices. Use `navigator.geolocation.getCurrentPosition()` to request the user's current position and use `navigator.geolocation.watchPosition()` to register a callback to be called when the user's position changes.
- The `navigator.vibrate()` method causes a mobile device (but not iOS) to vibrate. Often this is only allowed in response to a user gesture, but calling this method will allow your app to provide silent feedback that a gesture has been recognized.
- The ScreenOrientation API enables a web application to query the current orientation of a mobile device screen and also to lock themselves to landscape or portrait orientation.
- The "devicemotion" and "deviceorientation" events on the window object report accelerometer and magnetometer data for the device, enabling you to determine how the device is accelerating and how the user is orienting it in space. (These events do work on iOS.)
- The Sensor API is not yet widely supported beyond Chrome on Android devices, but it enables JavaScript access to the full suite of mobile device sensors, including accelerometer, gyroscope, magnetometer, and ambient light sensor. These sensors enable JavaScript to determine which direction a user is facing or to detect when the user shakes their phone, for example.

15.15.9 Binary APIs

Typed arrays, ArrayBuffers, and the DataView class (all covered in §11.2) enable JavaScript to work with binary data. As described earlier in this chapter, the `fetch()` API enables JavaScript programs to load binary data over the network. Another source of binary data is files from the user's local filesystem. For security reasons, JavaScript can't just read local files. But if the user selects a file for upload (using an `<input type="file">` form element) or uses drag-and-drop to drop a file into your web application, then JavaScript can access that file as a File object.

File is a subclass of Blob, and as such, it is an opaque representation of a chunk of data. You can use a FileReader class to asynchronously get the content of a file as an ArrayBuffer or string. (In some browsers, you can skip the FileReader and instead use the Promise-based `text()` and `arrayBuffer()` methods defined by the Blob class, or the `stream()` method for streaming access to the file contents.)

When working with binary data, especially streaming binary data, you may need to decode bytes into text or encode text as bytes. The TextEncoder and TextDecoder classes help with this task.

15.15.10 Media APIs

The `navigator.mediaDevices.getUserMedia()` function allows JavaScript to request access to the user's microphone and/or video camera. A successful request results in a MediaStream object. Video streams can be displayed in a `<video>` tag (by setting the `srcObject` property to the stream). Still frames of the video can be captured into an offscreen `<canvas>` with the canvas `drawImage()` function resulting in a relatively low-resolution photograph. Audio and video streams returned by `getUserMedia()` can be recorded and encoded to a Blob with a MediaRecorder object.

The more complex WebRTC API enables the transmission and reception of MediaStreams over the network, enabling peer-to-peer video conferencing, for example.

15.15.11 Cryptography and Related APIs

The `crypto` property of the Window object exposes a `getRandomValues()` method for cryptographically secure pseudorandom numbers. Other methods for encryption, decryption, key generation, digital signatures, and so on are available through `crypto.subtle`. The name of this property is a warning to everyone who uses these methods that properly using cryptographic algorithms is difficult and that you should not use those methods unless you really know what you are doing. Also, the methods of `crypto.subtle` are only available to JavaScript code running within documents that were loaded over a secure HTTPS connection.

The Credential Management API and the Web Authentication API allow JavaScript to generate, store, and retrieve public key (and other types of) credentials and enables account creation and login without passwords. The JavaScript API consists primarily of the functions `navigator.credentials.create()` and `navigator.credentials.get()`, but substantial infrastructure is required on the server side to make these methods work. These APIs are not universally supported yet, but have the potential to revolutionize the way we log in to websites.

The Payment Request API adds browser support for making credit card payments on the web. It allows users to store their payment details securely in the browser so that they don't have to type their credit card number each time they make a purchase. Web applications that want to request a payment create a PaymentRequest object and call its `show()` method to display the request to the user.

CHAPTER 16

Server-Side JavaScript with Node

Node is JavaScript with bindings to the underlying operating system, making it possible to write JavaScript programs that read and write files, execute child processes, and communicate over the network. This makes Node useful as a:

- Modern alternative to shell scripts that does not suffer from the arcane syntax of bash and other Unix shells.
- General-purpose programming language for running trusted programs, not subject to the security constraints imposed by web browsers on untrusted code.
- Popular environment for writing efficient and highly concurrent web servers.

The defining feature of Node is its single-threaded event-based concurrency enabled by an asynchronous-by-default API. If you have programmed in other languages but have not done much JavaScript coding, or if you're an experienced client-side JavaScript programmer used to writing code for web browers, using Node will be a bit of an adjustment, as is any new programming language or environment. This chapter begins by explaining the Node programming model, with an emphasis on concurrency, Node's API for working with streaming data, and Node's Buffer type for working with binary data. These initial sections are followed by sections that highlight and demonstrate some of the most important Node APIs, including those for working with files, networks, processes, and threads.

One chapter is not enough to document all of Node's APIs, but my hope is that this chapter will explain enough of the fundamentals to make you productive with Node, and confident that you can master any new APIs you need.

Installing Node

Node is open source software. Visit *https://nodejs.org* to download and install Node for Windows and MacOS. On Linux, you may be able to install Node with your normal package manager, or you can visit *https://nodejs.org/en/download* to download the binaries directly. If you work on containerized software, you can find official Node Docker images at *https://hub.docker.com*.

In addition to the Node executable, a Node installation also includes npm, a package manager that enables easy access to a vast ecosystem of JavaScript tools and libraries. The examples in this chapter will use only Node's built-in packages and will not require npm or any external libraries.

Finally, do not overlook the official Node documentation, available at *https://nodejs.org/api* and *https://nodejs.org/docs/guides*. I have found it to be well organized and well written.

16.1 Node Programming Basics

We'll begin this chapter with a quick look at how Node programs are structured and how they interact with the operating system.

16.1.1 Console Output

If you are used to JavaScript programming for web browsers, one of the minor surprises about Node is that `console.log()` is not just for debugging, but is Node's easiest way to display a message to the user, or, more generally, to send output to the stdout stream. Here's the classic "Hello World" program in Node:

```
console.log("Hello World!");
```

There are lower-level ways to write to stdout, but no fancier or more official way than simply calling `console.log()`.

In web browsers, `console.log()`, `console.warn()`, and `console.error()` typically display little icons next to their output in the developer console to indicate the variety of the log message. Node does not do this, but output displayed with `console.error()` is distinguished from output displayed with `console.log()` because `console.error()` writes to the stderr stream. If you're using Node to write a program that is designed to have stdout redirected to a file or a pipe, you can use `console.error()` to display text to the console where the user will see it, even though text printed with `console.log()` is hidden.

16.1.2 Command-Line Arguments and Environment Variables

If you have previously written Unix-style programs designed to be invoked from a terminal or other command-line interface, you know that these programs typically get their input primarily from command-line arguments and secondarily from environment variables.

Node follows these Unix conventions. A Node program can read its command-line arguments from the array of strings `process.argv`. The first element of this array is always the path to the Node executable. The second argument is the path to the file of JavaScript code that Node is executing. Any remaining elements in this array are the space-separated arguments that you passed on the command-line when you invoked Node.

For example, suppose you save this very short Node program to the file *argv.js*:

```
console.log(process.argv);
```

You can then execute the program and see output like this:

```
$ node --trace-uncaught argv.js --arg1 --arg2 filename
[
  '/usr/local/bin/node',
  '/private/tmp/argv.js',
  '--arg1',
  '--arg2',
  'filename'
]
```

There are a couple of things to note here:

- The first and second elements of `process.argv` will be fully qualified filesystem paths to the Node executable and the file of JavaScript that is being executed, even if you did not type them that way.
- Command-line arguments that are intended for and interpreted by the Node executable itself are consumed by the Node executable and do not appear in `process.argv`. (The `--trace-uncaught` command-line argument isn't actually doing anything useful in the previous example; it is just there to demonstrate that it does not appear in the output.) Any arguments (such as `--arg1` and `filename`) that appear after the name of the JavaScript file will appear in `process.argv`.

Node programs can also take input from Unix-style environment variables. Node makes these available though the `process.env` object. The property names of this object are environment variable names, and the property values (always strings) are the values of those variables.

Here is a partial list of environment variables on my system:

```
$ node -p -e 'process.env'
{
  SHELL: '/bin/bash',
  USER: 'david',
  PATH: '/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin',
  PWD: '/tmp',
  LANG: 'en_US.UTF-8',
  HOME: '/Users/david',
}
```

You can use `node -h` or `node --help` to find out what the `-p` and `-e` command-line arguments do. However, as a hint, note that you could rewrite the line above as `node --eval 'process.env' --print`.

16.1.3 Program Life Cycle

The `node` command expects a command-line argument that specifies the file of JavaScript code to be run. This initial file typically imports other modules of JavaScript code, and may also define its own classes and functions. Fundamentally, however, Node executes the JavaScript code in the specified file from top to bottom. Some Node programs exit when they are done executing the last line of code in the file. Often, however, a Node program will keep running long after the initial file has been executed. As we'll discuss in the following sections, Node programs are often asynchronous and based on callbacks and event handlers. Node programs do not exit until they are done running the initial file and until all callbacks have been called and there are no more pending events. A Node-based server program that listens for incoming network connections will theoretically run forever because it will always be waiting for more events.

A program can force itself to exit by calling `process.exit()`. Users can usually terminate a Node program by typing Ctrl-C in the terminal window where the program is running. A program can ignore Ctrl-C by registering a signal handler function with `process.on("SIGINT", ()=>{})`.

If code in your program throws an exception and no `catch` clause catches it, the program will print a stack trace and exit. Because of Node's asynchronous nature, exceptions that occur in callbacks or event handlers must be handled locally or not handled at all, which means that handling exceptions that occur in the asynchronous parts of your program can be a difficult problem. If you don't want these exceptions to cause your program to completely crash, register a global handler function that will be invoked instead of crashing:

```
process.setUncaughtExceptionCaptureCallback(e => {
    console.error("Uncaught exception:", e);
});
```

A similar situation arises if a Promise created by your program is rejected and there is no `.catch()` invocation to handle it. As of Node 13, this is not a fatal error that causes your program to exit, but it does print a verbose error message to the console. In some future version of Node, unhandled Promise rejections are expected to become fatal errors. If you do not want unhandled rejections, to print error messages or terminate your program, register a global handler function:

```
process.on("unhandledRejection", (reason, promise) => {
    // reason is whatever value would have been passed to a .catch() function
    // promise is the Promise object that rejected
});
```

16.1.4 Node Modules

Chapter 10 documented JavaScript module systems, covering both Node modules and ES6 modules. Because Node was created before JavaScript had a module system, Node had to create its own. Node's module system uses the `require()` function to import values into a module and the `exports` object or the `module.exports` property to export values from a module. These are a fundamental part of the Node programming model, and they are covered in detail in §10.2.

Node 13 adds support for standard ES6 modules as well as require-based modules (which Node calls "CommonJS modules"). The two module systems are not fully compatible, so this is somewhat tricky to do. Node needs to know—before it loads a module—whether that module will be using `require()` and `module.exports` or if it will be using `import` and `export`. When Node loads a file of JavaScript code as a CommonJS module, it automatically defines the `require()` function along with identifiers `exports` and `module`, and it does not enable the `import` and `export` keywords. On the other hand, when Node loads a file of code as an ES6 module, it must enable the `import` and `export` declarations, and it must *not* define extra identifiers like `require`, `module`, and `exports`.

The simplest way to tell Node what kind of module it is loading is to encode this information in the file extension. If you save your JavaScript code in a file that ends with *.mjs*, then Node will always load it as an ES6 module, will expect it to use `import` and `export`, and will not provide a `require()` function. And if you save your code in a file that ends with *.cjs*, then Node will always treat it as a CommonJS module, will provide a `require()` function, and will throw a SyntaxError if you use `import` or `export` declarations.

For files that do not have an explicit *.mjs* or *.cjs* extension, Node looks for a file named *package.json* in the same directory as the file and then in each of the containing directories. Once the nearest *package.json* file is found, Node checks for a top-level `type` property in the JSON object. If the value of the `type` property is "module", then Node loads the file as an ES6 module. If the value of that property is

"commonjs", then Node loads the file as a CommonJS module. Note that you do not need to have a *package.json* file to run Node programs: when no such file is found (or when the file is found but it does not have a `type` property), Node defaults to using CommonJS modules. This *package.json* trick only becomes necessary if you want to use ES6 modules with Node and do not want to use the *.mjs* file extension.

Because there is an enormous amount of existing Node code written using CommonJS module format, Node allows ES6 modules to load CommonJS modules using the `import` keyword. The reverse is not true, however: a CommonJS module cannot use `require()` to load an ES6 module.

16.1.5 The Node Package Manager

When you install Node, you typically get a program named npm as well. This is the Node Package Manager, and it helps you download and manage libraries that your program depends on. npm keeps track of those dependencies (as well as other information about your program) in a file named *package.json* in the root directory of your project. This *package.json* file created by npm is where you would add `"type":"module"` if you wanted to use ES6 modules for your project.

This chapter does not cover npm in any detail (but see §17.4 for a little more depth). I'm mentioning it here because unless you write programs that do not use any external libraries, you will almost certainly be using npm or a tool like it. Suppose, for example, that you are going to be developing a web server and plan to use the Express framework (*https://expressjs.com*) to simplify the task. To get started, you might create a directory for your project, and then, in that directory type `npm init`. npm will ask you for your project name, version number, etc., and will then create an initial *package.json* file based on your responses.

Now to start using Express, you'd type `npm install express`. This tells npm to download the Express library along with all of its dependencies and install all the packages in a local *node_modules/* directory:

```
$ npm install express
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN my-server@1.0.0 No description
npm WARN my-server@1.0.0 No repository field.

+ express@4.17.1
added 50 packages from 37 contributors and audited 126 packages in 3.058s
found 0 vulnerabilities
```

When you install a package with npm, npm records this dependency—that your project depends on Express—in the *package.json* file. With this dependency recorded in *package.json*, you could give another programmer a copy of your code and your

package.json, and they could simply type `npm install` to automatically download and install all of the libraries that your program needs in order to run.

16.2 Node Is Asynchronous by Default

JavaScript is a general-purpose programming language, so it is perfectly possible to write CPU-intensive programs that multiply large matrices or perform complicated statistical analyses. But Node was designed and optimized for programs—like network servers—that are I/O intensive. And in particular, Node was designed to make it possible to easily implement highly concurrent servers that can handle many requests at the same time.

Unlike many programming languages, however, Node does not achieve concurrency with threads. Multithreaded programming is notoriously hard to do correctly, and difficult to debug. Also, threads are a relatively heavyweight abstraction and if you want to write a server that can handle hundreds of concurrent requests, using hundreds of threads may require a prohibitive amount of memory. So Node adopts the single-threaded JavaScript programming model that the web uses, and this turns out to be a vast simplification that makes the creation of network servers a routine skill rather than an arcane one.

True Parallelism with Node

Node programs can run multiple operating system processes, and Node 10 and later support Worker objects (§16.11), which are a kind of thread borrowed from web browsers. If you use multiple processes or create one or more Worker threads and run your program on a system with more than one CPU, then your program will no longer be single-threaded and your program will truly be executing multiple streams of code in parallel. These techniques can be valuable for CPU-intensive operations but are not commonly used for I/O-intensive programs like servers.

It is worth noting, however, that Node's processes and Workers avoid the typical complexity of multithreaded programming because interprocess and inter-Worker communication is via message passing and they cannot easily share memory with each other.

Node achieves high levels of concurrency while maintaining a single-threaded programming model by making its API asynchronous and nonblocking by default. Node takes its nonblocking approach very seriously and to an extreme that may surprise you. You probably expect functions that read from and write to the network to be asynchronous, but Node goes further and defines nonblocking asynchronous functions for reading and writing files from the local filesystem. This makes sense, when you think about it: the Node API was designed in the days when spinning hard drives

were still the norm and there really were milliseconds of blocking "seek time" while waiting for the disc to spin around before a file operation could begin. And in modern datacenters, the "local" filesystem may actually be across the network somewhere with network latencies on top of drive latencies. But even if reading a file asynchronously seems normal to you, Node takes it still further: the default functions for initiating a network connection or looking up a file modification time, for example, are also nonblocking.

Some functions in Node's API are synchronous but nonblocking: they run to completion and return without ever needing to block. But most of the interesting functions perform some kind of input or output, and these are asynchronous functions so they can avoid even the tiniest amount of blocking. Node was created before JavaScript had a Promise class, so asynchronous Node APIs are callback-based. (If you have not yet read or have already forgotten Chapter 13, this would be a good time to skip back to that chapter.) Generally, the last argument you pass to an asynchronous Node function is a callback. Node uses *error-first callbacks*, which are typically invoked with two arguments. The first argument to an error-first callback is normally `null` in the case where no error occurred, and the second argument is whatever data or response was produced by the original asynchronous function you called. The reason for putting the error argument first is to make it impossible for you to omit it, and you should always check for a non-null value in this argument. If it is an Error object, or even an integer error code or string error message, then something went wrong. In this case, the second argument to your callback function is likely to be `null`.

The following code demonstrates how to use the nonblocking `readFile()` function to read a configuration file, parse it as JSON, and then pass the parsed configuration object to another callback:

```
const fs = require("fs");  // Require the filesystem module

// Read a config file, parse its contents as JSON, and pass the
// resulting value to the callback. If anything goes wrong,
// print an error message to stderr and invoke the callback with null
function readConfigFile(path, callback) {
    fs.readFile(path, "utf8", (err, text) => {
        if (err) {    // Something went wrong reading the file
            console.error(err);
            callback(null);
            return;
        }
        let data = null;
        try {
            data = JSON.parse(text);
        } catch(e) {  // Something went wrong parsing the file contents
            console.error(e);
        }
        callback(data);
```

```
    });
}
```

Node predates standardized promises, but because it is fairly consistent about its error-first callbacks, it is easy to create Promise-based variants of its callback-based APIs using the `util.promisify()` wrapper. Here's how we could rewrite the `readConfigFile()` function to return a Promise:

```
const util = require("util");
const fs = require("fs");  // Require the filesystem module
const pfs = {              // Promise-based variants of some fs functions
    readFile: util.promisify(fs.readFile)
};

function readConfigFile(path) {
    return pfs.readFile(path, "utf-8").then(text => {
        return JSON.parse(text);
    });
}
```

We can also simpify the preceding Promise-based function using `async` and `await` (again, if you have not yet read through Chapter 13, this would be a good time to do so):

```
async function readConfigFile(path) {
    let text = await pfs.readFile(path, "utf-8");
    return JSON.parse(text);
}
```

The `util.promisify()` wrapper can produce a Promise-based version of many Node functions. In Node 10 and later, the `fs.promises` object has a number of predefined Promise-based functions for working with the filesystem. We'll discuss them later in this chapter, but note that in the preceding code, we could replace `pfs.readFile()` with `fs.promises.readFile()`.

We had said that Node's programming model is async-by-default. But for programmer convenience, Node does define blocking, synchronous variants of many of its functions, especially in the filesystem module. These functions typically have names that are clearly labeled with `Sync` at the end.

When a server is first starting up and is reading its configuration files, it is not handling network requests yet, and little or no concurrency is actually possible. So in this situation, there is really no need to avoid blocking, and we can safely use blocking functions like `fs.readFileSync()`. We can drop the `async` and `await` from this code and write a purely synchronous version of our `readConfigFile()` function. Instead of invoking a callback or returning a Promise, this function simply returns the parsed JSON value or throws an exception:

```
const fs = require("fs");
function readConfigFileSync(path) {
```

```
    let text = fs.readFileSync(path, "utf-8");
    return JSON.parse(text);
}
```

In addition to its error-first two-argument callbacks, Node also has a number of APIs that use event-based asynchrony, typically for handling streaming data. We'll cover Node events in more detail later.

Now that we've discussed Node's aggressively nonblocking API, let's turn back to the topic of concurrency. Node's built-in nonblocking functions work using the operating system's version of callbacks and event handlers. When you call one of these functions, Node takes action to get the operation started, then registers some kind of event handler with the operating system so that it will be notified when the operation is complete. The callback you passed to the Node function gets stored internally so that Node can invoke your callback when the operating system sends the appropriate event to Node.

This kind of concurrency is often called event-based concurrency. At its core, Node has a single thread that runs an "event loop." When a Node program starts, it runs whatever code you've told it to run. This code presumably calls at least one nonblocking function causing a callback or event handler to be registered with the operating system. (If not, then you've written a synchronous Node program, and Node simply exits when it reaches the end.) When Node reaches the end of your program, it blocks until an event happens, at which time the OS starts it running again. Node maps the OS event to the JavaScript callback you registered and then invokes that function. Your callback function may invoke more nonblocking Node functions, causing more OS event handlers to be registered. Once your callback function is done running, Node goes back to sleep again and the cycle repeats.

For web servers and other I/O-intensive applications that spend most of their time waiting for input and output, this style of event-based concurrency is efficient and effective. A web server can concurrently handle requests from 50 different clients without needing 50 different threads as long as it uses nonblocking APIs and there is some kind of internal mapping from network sockets to JavaScript functions to invoke when activity occurs on those sockets.

16.3 Buffers

One of the datatypes you're likely to use frequently in Node—especially when reading data from files or from the network—is the Buffer class. A Buffer is a lot like a string, except that it is a sequence of bytes instead of a sequence of characters. Node was created before core JavaScript supported typed arrays (see §11.2) and there was no Uint8Array to represent an array of unsigned bytes. Node defined the Buffer class to fill that need. Now that Uint8Array is part of the JavaScript language, Node's Buffer class is a subclass of Uint8Array.

What distinguishes Buffer from its Uint8Array superclass is that it is designed to interoperate with JavaScript strings: the bytes in a buffer can be initialized from character strings or converted to character strings. A character encoding maps each character in some set of characters to an integer. Given a string of text and a character encoding, we can *encode* the characters in the string into a sequence of bytes. And given a (properly encoded) sequence of bytes and a character encoding, we can *decode* those bytes into a sequence of characters. Node's Buffer class has methods that perform both encoding and decoding, and you can recognize these methods because they expect an `encoding` argument that specifies the encoding to be used.

Encodings in Node are specified by name, as strings. The supported encodings are:

`"utf8"`
: This is the default when no encoding is specified, and is the Unicode encoding you are most likely to use.

`"utf16le"`
: Two-byte Unicode characters, with little-endian ordering. Codepoints above `\uffff` are encoded as a pair of two-byte sequences. Encoding `"ucs2"` is an alias.

`"latin1"`
: The one-byte-per-character ISO-8859-1 encoding that defines a character set suitable for many Western European languages. Because there is a one-to-one mapping between bytes and latin-1 characters, this encoding is also known as `"binary"`.

`"ascii"`
: The 7-bit English-only ASCII encoding, a strict subset of the `"utf8"` encoding.

`"hex"`
: This encoding converts each byte to a pair of ASCII hexadecimal digits.

`"base64"`
: This encoding converts each sequence of three bytes into a sequence of four ascii characters.

Here is some example code that demonstrates how to work with Buffers and how to convert to and from strings:

```
let b = Buffer.from([0x41, 0x42, 0x43]);          // <Buffer 41 42 43>
b.toString()                                      // => "ABC"; default "utf8"
b.toString("hex")                                 // => "414243"

let computer = Buffer.from("IBM3111", "ascii");   // Convert string to Buffer
for(let i = 0; i < computer.length; i++) {        // Use Buffer as byte array
    computer[i]--;                                // Buffers are mutable
}
computer.toString("ascii")                        // => "HAL2000"
```

```
computer.subarray(0,3).map(x=>x+1).toString()     // => "IBM"

// Create new "empty" buffers with Buffer.alloc()
let zeros = Buffer.alloc(1024);                   // 1024 zeros
let ones = Buffer.alloc(128, 1);                  // 128 ones
let dead = Buffer.alloc(1024, "DEADBEEF", "hex"); // Repeating pattern of bytes

// Buffers have methods for reading and writing multi-byte values
// from and to a buffer at any specified offset.
dead.readUInt32BE(0)       // => 0xDEADBEEF
dead.readUInt32BE(1)       // => 0xADBEEFDE
dead.readBigUInt64BE(6)    // => 0xBEEFDEADBEEFDEADn
dead.readUInt32LE(1020)    // => 0xEFBEADDE
```

If you write a Node program that actually manipulates binary data, you may find yourself using the Buffer class extensively. On the other hand, if you are just working with text that is read from or written to a file or the network, then you may only encounter Buffer as an intermediate representation of your data. A number of Node APIs can take input or return output as either strings or Buffer objects. Typically, if you pass a string, or expect a string to be returned, from one of these APIs, you'll need to specify the name of the text encoding you want to use. And if you do this, then you may not need to use a Buffer object at all.

16.4 Events and EventEmitter

As described, all of Node's APIs are asynchronous by default. For many of them, this asynchrony takes the form of two-argument error-first callbacks that are invoked when the requested operation is complete. But some of the more complicated APIs are event-based instead. This is typically the case when the API is designed around an object rather than a function, or when a callback function needs to be invoked multiple times, or when there are multiple types of callback functions that may be required. Consider the `net.Server` class, for example: an object of this type is a server socket that is used to accept incoming connections from clients. It emits a "listening" event when it first starts listening for connections, a "connection" event every time a client connects, and a "close" event when it has been closed and is no longer listening.

In Node, objects that emit events are instances of EventEmitter or a subclass of EventEmitter:

```
const EventEmitter = require("events"); // Module name does not match class name
const net = require("net");
let server = new net.Server();          // create a Server object
server instanceof EventEmitter          // => true: Servers are EventEmitters
```

The main feature of EventEmitters is that they allow you to register event handler functions with the `on()` method. EventEmitters can emit multiple types of events, and event types are identified by name. To register an event handler, call the `on()` method,

passing the name of the event type and the function that should be invoked when an event of that type occurs. EventEmitters can invoke handler functions with any number of arguments, and you need to read the documentation for a specific kind of event from a specific EventEmitter to know what arguments you should expect to be passed:

```
const net = require("net");
let server = new net.Server();          // create a Server object
server.on("connection", socket => {     // Listen for "connection" events
    // Server "connection" events are passed a socket object
    // for the client that just connected. Here we send some data
    // to the client and disconnect.
    socket.end("Hello World", "utf8");
});
```

If you prefer more explicit method names for registering event listeners, you can also use `addListener()`. And you can remove a previously registered event listener with `off()` or `removeListener()`. As a special case, you can register an event listener that will be automatically removed after it is triggered for the first time by calling `once()` instead of `on()`.

When an event of a particular type occurs for a particular EventEmitter object, Node invokes all of the handler functions that are currently registered on that EventEmitter for events of that type. They are invoked in order from the first registered to the last registered. If there is more than one handler function, they are invoked sequentially on a single thread: there is no parallelism in Node, remember. And, importantly, event handling functions are invoked synchronously, not asynchronously. What this means is that the `emit()` method does not queue up event handlers to be invoked at some later time. `emit()` invokes all the registered handlers, one after the other, and does not return until the last event handler has returned.

What this means, in effect, is that when one of the built-in Node APIs emits an event, that API is basically blocking on your event handlers. If you write an event handler that calls a blocking function like `fs.readFileSync()`, no further event handling will happen until your synchronous file read is complete. If your program is one—like a network server—that needs to be responsive, then it is important that you keep your event handler functions nonblocking and fast. If you need to do a lot of computation when an event occurs, it is often best to use the handler to schedule that computation asynchronously using `setTimeout()` (see §11.10). Node also defines `setImmediate()`, which schedules a function to be invoked immediately after all pending callbacks and events have been handled.

The EventEmitter class also defines an `emit()` method that causes the registered event handler functions to be invoked. This is useful if you are defining your own event-based API, but is not commonly used when you're just programming with existing APIs. `emit()` must be invoked with the name of the event type as its first

argument. Any additional arguments that are passed to `emit()` become arguments to the registered event handler functions. The handler functions are also invoked with the `this` value set to the EventEmitter object itself, which is often convenient. (Remember, though, that arrow functions always use the `this` value of the context in which they are defined, and they cannot be invoked with any other `this` value. Nevertheless, arrow functions are often the most convenient way to write event handlers.)

Any value returned by an event handler function is ignored. If an event handler function throws an exception, however, it propagates out from the `emit()` call and prevents the execution of any handler functions that were registered after the one that threw the exception.

Recall that Node's callback-based APIs use error-first callbacks, and it is important that you always check the first callback argument to see if an error occurred. With event-based APIs, the equivalent is "error" events. Since event-based APIs are often used for networking and other forms of streaming I/O, they are vulnerable to unpredictable asynchronous errors, and most EventEmitters define an "error" event that they emit when an error occurs. Whenever you use an event-based API, you should make it a habit to register a handler for "error" events. "Error" events get special treatment by the EventEmitter class. If `emit()` is called to emit an "error" event, and if there are no handlers registered for that event type, then an exception will be thrown. Since this occurs asynchronously, there is no way for you to handle the exception in a `catch` block, so this kind of error typically causes your program to exit.

16.5 Streams

When implementing an algorithm to process data, it is almost always easiest to read all the data into memory, do the processing, and then write the data out. For example, you could write a Node function to copy a file like this.[1]

```
const fs = require("fs");

// An asynchronous but nonstreaming (and therefore inefficient) function.
function copyFile(sourceFilename, destinationFilename, callback) {
    fs.readFile(sourceFilename, (err, buffer) => {
        if (err) {
            callback(err);
        } else {
            fs.writeFile(destinationFilename, buffer, callback);
        }
    });
}
```

1 Node defines a `fs.copyFile()` function that you would actually use in practice.

This `copyFile()` function uses asynchronous functions and callbacks, so it does not block and is suitable for use in concurrent programs like servers. But notice that it must allocate enough memory to hold the entire contents of the file in memory at once. This may be fine in some use cases, but it starts to fail if the files to be copied are very large, or if your program is highly concurrent and there may be many files being copied at the same time. Another shortcoming of this `copyFile()` implementation is that it cannot start writing the new file until it has finished reading the old file.

The solution to these problems is to use streaming algorithms where data "flows" into your program, is processed, and then flows out of your program. The idea is that your algorithm processes the data in small chunks and the full dataset is never held in memory at once. When streaming solutions are possible, they are more memory efficient and can also be faster. Node's networking APIs are stream-based and Node's filesystem module defines streaming APIs for reading and writing files, so you are likely to use a streaming API in many of the Node programs that you write. We'll see a streaming version of the `copyFile()` function in "Flowing mode" on page 598.

Node supports four basic stream types:

Readable
: Readable streams are sources of data. The stream returned by `fs.createReadStream()`, for example, is a stream from which the content of a specified file can be read. `process.stdin` is another Readable stream that returns data from standard input.

Writable
: Writable streams are sinks or destinations for data. The return value of `fs.createWriteStream()`, for example, is a Writable stream: it allows data to be written to it in chunks, and outputs all of that data to a specified file.

Duplex
: Duplex streams combine a Readable stream and a Writable stream into one object. The Socket objects returned by `net.connect()` and other Node networking APIs, for example, are Duplex streams. If you write to a socket, your data is sent across the network to whatever computer the socket is connected to. And if you read from a socket, you access the data written by that other computer.

Transform
: Transform streams are also readable and writable, but they differ from Duplex streams in an important way: data written to a Transform stream becomes readable—usually in some transformed form—from the same stream. The `zlib.createGzip()` function, for example, returns a Transform stream that compresses (with the *gzip* algorithm) the data written to it. In a similar way, the `crypto.createCipheriv()` function returns a Transform stream that encrypts or decrypts data that is written to it.

By default, streams read and write buffers. If you call the `setEncoding()` method of a Readable stream, it will return decoded strings to you instead of Buffer objects. And if you write a string to a Writable buffer, it will be automatically encoded using the buffer's default encoding or whatever encoding you specify. Node's stream API also supports an "object mode" where streams read and write objects more complex than buffers and strings. None of Node's core APIs use this object mode, but you may encounter it in other libraries.

Readable streams have to read their data from somewhere, and Writable streams have to write their data to somewhere, so every stream has two ends: an input and an output or a source and a destination. The tricky thing about stream-based APIs is that the two ends of the stream will almost always flow at different speeds. Perhaps the code that reads from a stream wants to read and process data more quickly than the data is actually being written into the stream. Or the reverse: perhaps data is written to a stream more quickly than it can be read and pulled out of the stream on the other end. Stream implementations almost always include an internal buffer to hold data that has been written but not yet read. Buffering helps to ensure that there is data available to read when it's requested, and that there is space to hold data when it is written. But neither of these things can ever be guaranteed, and it is the nature of stream-based programming that readers will sometimes have to wait for data to be written (because the stream buffer is empty), and writers will sometimes have to wait for data to be read (because the stream buffer is full).

In programming environments that use thread-based concurrency, stream APIs typically have blocking calls: a call to read data does not return until data arrives in the stream and a call to write data blocks until there is enough room in the stream's internal buffer to accommodate the new data. With an event-based concurrency model, however, blocking calls do not make sense, and Node's stream APIs are event- and callback-based. Unlike other Node APIs, there are not "Sync" versions of the methods that will be described later in this chapter.

The need to coordinate stream readability (buffer not empty) and writability (buffer not full) via events makes Node's stream APIs somewhat complicated. This is compounded by the fact that these APIs have evolved and changed over the years: for Readable streams, there are two completely distinct APIs that you can use. Despite the complexity, it is worth understanding and mastering Node's streaming APIs because they enable high-throughput I/O in your programs.

The subsections that follow demonstrate how to read and write from Node's stream classes.

16.5.1 Pipes

Sometimes, you need to read data from a stream simply to turn around and write that same data to another stream. Imagine, for example, that you are writing a simple

HTTP server that serves a directory of static files. In this case, you will need to read data from a file input stream and write it out to a network socket. But instead of writing your own code to handle the reading and writing, you can instead simply connect the two sockets together as a "pipe" and let Node handle the complexities for you. Simply pass the Writable stream to the `pipe()` method of the Readable stream:

```
const fs = require("fs");

function pipeFileToSocket(filename, socket) {
    fs.createReadStream(filename).pipe(socket);
}
```

The following utility function pipes one stream to another and invokes a callback when done or when an error occurs:

```
function pipe(readable, writable, callback) {
    // First, set up error handling
    function handleError(err) {
        readable.close();
        writable.close();
        callback(err);
    }

    // Next define the pipe and handle the normal termination case
    readable
        .on("error", handleError)
        .pipe(writable)
        .on("error", handleError)
        .on("finish", callback);
}
```

Transform streams are particularly useful with pipes, and create pipelines that involve more than two streams. Here's an example function that compresses a file:

```
const fs = require("fs");
const zlib = require("zlib");

function gzip(filename, callback) {
    // Create the streams
    let source = fs.createReadStream(filename);
    let destination = fs.createWriteStream(filename + ".gz");
    let gzipper = zlib.createGzip();

    // Set up the pipeline
    source
        .on("error", callback)   // call callback on read error
        .pipe(gzipper)
        .pipe(destination)
        .on("error", callback)   // call callback on write error
        .on("finish", callback); // call callback when writing is complete
}
```

Using the `pipe()` method to copy data from a Readable stream to a Writable stream is easy, but in practice, you often need to process the data somehow as it streams through your program. One way to do this is to implement your own Transform stream to do that processing, and this approach allows you to avoid manually reading and writing the streams. Here, for example, is a function that works like the Unix `grep` utility: it reads lines of text from an input stream, but writes only the lines that match a specified regular expression:

```
const stream = require("stream");

class GrepStream extends stream.Transform {
    constructor(pattern) {
        super({decodeStrings: false});// Don't convert strings back to buffers
        this.pattern = pattern;       // The regular expression we want to match
        this.incompleteLine = "";     // Any remnant of the last chunk of data
    }

    // This method is invoked when there is a string ready to be
    // transformed. It should pass transformed data to the specified
    // callback function. We expect string input so this stream should
    // only be connected to readable streams that have had
    // setEncoding() called on them.
    _transform(chunk, encoding, callback) {
        if (typeof chunk !== "string") {
            callback(new Error("Expected a string but got a buffer"));
            return;
        }
        // Add the chunk to any previously incomplete line and break
        // everything into lines
        let lines = (this.incompleteLine + chunk).split("\n");

        // The last element of the array is the new incomplete line
        this.incompleteLine = lines.pop();

        // Find all matching lines
        let output = lines                      // Start with all complete lines,
            .filter(l => this.pattern.test(l)) // filter them for matches,
            .join("\n");                        // and join them back up.

        // If anything matched, add a final newline
        if (output) {
            output += "\n";
        }

        // Always call the callback even if there is no output
        callback(null, output);
    }

    // This is called right before the stream is closed.
    // It is our chance to write out any last data.
    _flush(callback) {
```

```
            // If we still have an incomplete line, and it matches
            // pass it to the callback
            if (this.pattern.test(this.incompleteLine)) {
                callback(null, this.incompleteLine + "\n");
            }
        }
    }

    // Now we can write a program like 'grep' with this class.
    let pattern = new RegExp(process.argv[2]); // Get a RegExp from command line.
    process.stdin                              // Start with standard input,
        .setEncoding("utf8")                   // read it as Unicode strings,
        .pipe(new GrepStream(pattern))         // pipe it to our GrepStream,
        .pipe(process.stdout)                  // and pipe that to standard out.
        .on("error", () => process.exit());    // Exit gracefully if stdout closes.
```

16.5.2 Asynchronous Iteration

In Node 12 and later, Readable streams are asynchronous iterators, which means that within an `async` function you can use a `for/await` loop to read string or Buffer chunks from a stream using code that is structured like synchronous code would be. (See §13.4 for more on asynchronous iterators and `for/await` loops.)

Using an asynchronous iterator is almost as easy as using the `pipe()` method, and is probably easier when you need to process each chunk you read in some way. Here's how we could rewrite the `grep` program in the previous section using an `async` function and a `for/await` loop:

```
// Read lines of text from the source stream, and write any lines
// that match the specified pattern to the destination stream.
async function grep(source, destination, pattern, encoding="utf8") {
    // Set up the source stream for reading strings, not Buffers
    source.setEncoding(encoding);

    // Set an error handler on the destination stream in case standard
    // output closes unexpectedly (when piping output to `head`, e.g.)
    destination.on("error", err => process.exit());

    // The chunks we read are unlikely to end with a newline, so each will
    // probably have a partial line at the end. Track that here
    let incompleteLine = "";

    // Use a for/await loop to asynchronously read chunks from the input stream
    for await (let chunk of source) {
        // Split the end of the last chunk plus this one into lines
        let lines = (incompleteLine + chunk).split("\n");
        // The last line is incomplete
        incompleteLine = lines.pop();
        // Now loop through the lines and write any matches to the destination
        for(let line of lines) {
            if (pattern.test(line)) {
```

```
                destination.write(line + "\n", encoding);
            }
        }
    }
    // Finally, check for a match on any trailing text.
    if (pattern.test(incompleteLine)) {
        destination.write(incompleteLine + "\n", encoding);
    }
}

let pattern = new RegExp(process.argv[2]);   // Get a RegExp from command line.
grep(process.stdin, process.stdout, pattern) // Call the async grep() function.
    .catch(err => {                          // Handle asynchronous exceptions.
        console.error(err);
        process.exit();
    });
```

16.5.3 Writing to Streams and Handling Backpressure

The async `grep()` function in the preceding code example demonstrated how to use a Readable stream as an asynchronous iterator, but it also demonstrated that you can write data to a Writable stream simply by passing it to the `write()` method. The `write()` method takes a buffer or string as the first argument. (Object streams expect other kinds of objects, but are beyond the scope of this chapter.) If you pass a buffer, the bytes of that buffer will be written directly. If you pass a string, it will be encoded to a buffer of bytes before being written. Writable streams have a default encoding that is used when you pass a string as the only argument to `write()`. The default encoding is typically "utf8," but you can set it explicitly by calling `setDefaultEncoding()` on the Writable stream. Alternatively, when you pass a string as the first argument to `write()` you can pass an encoding name as the second argument.

`write()` optionally takes a callback function as its third argument. This will be invoked when the data has actually been written and is no longer in the Writable stream's internal buffer. (This callback may also be invoked if an error occurs, but this is not guaranteed. You should register an "error" event handler on the Writable stream to detect errors.)

The `write()` method has a very important return value. When you call `write()` on a stream, it will always accept and buffer the chunk of data you have passed. It then returns `true` if the internal buffer is not yet full. Or, if the buffer is now full or overfull, it returns `false`. This return value is advisory, and you can ignore it—Writable streams will enlarge their internal buffer as much as needed if you keep calling `write()`. But remember that the reason to use a streaming API in the first place is to avoid the cost of keeping lots of data in memory at once.

A return value of `false` from the `write()` method is a form of *backpressure*: a message from the stream that you have written data more quickly than it can be handled.

The proper response to this kind of backpressure is to stop calling `write()` until the stream emits a "drain" event, signaling that there is once again room in the buffer. Here, for example, is a function that writes to a stream, and then invokes a callback when it is OK to write more data to the stream:

```
function write(stream, chunk, callback) {
    // Write the specified chunk to the specified stream
    let hasMoreRoom = stream.write(chunk);

    // Check the return value of the write() method:
    if (hasMoreRoom) {                  // If it returned true, then
        setImmediate(callback);         // invoke callback asynchronously.
    } else {                            // If it returned false, then
        stream.once("drain", callback); // invoke callback on drain event.
    }
}
```

The fact that it is sometimes OK to call `write()` multiple times in a row and sometimes you have to wait for an event between writes makes for awkward algorithms. This is one of the reasons that using the `pipe()` method is so appealing: when you use `pipe()`, Node handles backpressure for you automatically.

If you are using `await` and `async` in your program, and are treating Readable streams as asynchronous iterators, it is straightforward to implement a Promise-based version of the `write()` utility function above to properly handle backpressure. In the async `grep()` function we just looked at, we did not handle backpressure. The async `copy()` function in the following example demonstrates how it can be done correctly. Note that this function just copies chunks from a source stream to a destination stream and calling `copy(source, destination)` is much like calling `source.pipe(destination)`:

```
// This function writes the specified chunk to the specified stream and
// returns a Promise that will be fulfilled when it is OK to write again.
// Because it returns a Promise, it can be used with await.
function write(stream, chunk) {
    // Write the specified chunk to the specified stream
    let hasMoreRoom = stream.write(chunk);

    if (hasMoreRoom) {                    // If buffer is not full, return
        return Promise.resolve(null);     // an already resolved Promise object
    } else {
        return new Promise(resolve => {   // Otherwise, return a Promise that
            stream.once("drain", resolve); // resolves on the drain event.
        });
    }
}

// Copy data from the source stream to the destination stream
// respecting backpressure from the destination stream.
// This is much like calling source.pipe(destination).
```

```
async function copy(source, destination) {
    // Set an error handler on the destination stream in case standard
    // output closes unexpectedly (when piping output to `head`, e.g.)
    destination.on("error", err => process.exit());

    // Use a for/await loop to asynchronously read chunks from the input stream
    for await (let chunk of source) {
        // Write the chunk and wait until there is more room in the buffer.
        await write(destination, chunk);
    }
}

// Copy standard input to standard output
copy(process.stdin, process.stdout);
```

Before we conclude this discussion of writing to streams, note again that failing to respond to backpressure can cause your program to use more memory than it should when the internal buffer of a Writable stream overflows and grows larger and larger. If you are writing a network server, this can be a remotely exploitable security issue. Suppose you write an HTTP server that delivers files over the network, but you didn't use `pipe()` and you didn't take the time to handle backpressure from the `write()` method. An attacker could write an HTTP client that initiates requests for large files (such as images) but never actually reads the body of the request. Since the client is not reading the data over the network, and the server isn't responding to backpressure, buffers on the server are going to overflow. With enough concurrent connections from the attacker, this can turn into a denial-of-service attack that slows your server down or even crashes it.

16.5.4 Reading Streams with Events

Node's readable streams have two modes, each of which has its own API for reading. If you can't use pipes or asynchronous iteration in your program, you will need to pick one of these two event-based APIs for handling streams. It is important that you use only one or the other and do not mix the two APIs.

Flowing mode

In *flowing mode*, when readable data arrives, it is immediately emitted in the form of a "data" event. To read from a stream in this mode, simply register an event handler for "data" events, and the stream will push chunks of data (buffers or strings) to you as soon as they becomes available. Note that there is no need to call the `read()` method in flowing mode: you only need to handle "data" events. Note that newly created streams do not start off in flowing mode. Registering a "data" event handler switches a stream into flowing mode. Conveniently, this means that a stream does not emit "data" events until you register the first "data" event handler.

If you are using flowing mode to read data from a Readable stream, process it, then write it to a Writable stream, then you may need to handle backpressure from the Writable stream. If the `write()` method returns `false` to indicate that the write buffer is full, you can call `pause()` on the Readable stream to temporarily stop `data` events. Then, when you get a "drain" event from the Writable stream, you can call `resume()` on the Readable stream to start the "data" events flowing again.

A stream in flowing mode emits an "end" event when the end of the stream is reached. This event indicates that no more "data" events will ever be emitted. And, as with all streams, an "error" event is emitted if an error occurs.

At the beginning of this section on streams, we showed a nonstreaming `copyFile()` function and promised a better version to come. The following code shows how to implement a streaming `copyFile()` function that uses the flowing mode API and handles backpressure. This would have been easier to implement with a `pipe()` call, but it serves here as a useful demonstration of the multiple event handlers that are used to coordinate data flow from one stream to the other.

```
const fs = require("fs");

// A streaming file copy function, using "flowing mode".
// Copies the contents of the named source file to the named destination file.
// On success, invokes the callback with a null argument. On error,
// invokes the callback with an Error object.
function copyFile(sourceFilename, destinationFilename, callback) {
    let input = fs.createReadStream(sourceFilename);
    let output = fs.createWriteStream(destinationFilename);

    input.on("data", (chunk) => {          // When we get new data,
        let hasRoom = output.write(chunk); // write it to the output stream.
        if (!hasRoom) {                    // If the output stream is full
            input.pause();                 // then pause the input stream.
        }
    });
    input.on("end", () => {                // When we reach the end of input,
        output.end();                      // tell the output stream to end.
    });
    input.on("error", err => {             // If we get an error on the input,
        callback(err);                     // call the callback with the error
        process.exit();                    // and quit.
    });

    output.on("drain", () => {             // When the output is no longer full,
        input.resume();                    // resume data events on the input
    });
    output.on("error", err => {            // If we get an error on the output,
        callback(err);                     // call the callback with the error
        process.exit();                    // and quit.
    });
    output.on("finish", () => {            // When output is fully written
```

```
            callback(null);                    // call the callback with no error.
        });
    }

    // Here's a simple command-line utility to copy files
    let from = process.argv[2], to = process.argv[3];
    console.log(`Copying file ${from} to ${to}...`);
    copyFile(from, to, err => {
        if (err) {
            console.error(err);
        } else {
            console.log("done.");
        }
    });
```

Paused mode

The other mode for Readable streams is "paused mode." This is the mode that streams start in. If you never register a "data" event handler and never call the `pipe()` method, then a Readable stream remains in paused mode. In paused mode, the stream does not push data to you in the form of "data" events. Instead, you pull data from the stream by explicitly calling its `read()` method. This is not a blocking call, and if there is no data available to read on the stream, it will return `null`. Since there is not a synchronous API to wait for data, the paused mode API is also event-based. A Readable stream in paused mode emits "readable" events when data becomes available to read on the stream. In response, your code should call the `read()` method to read that data. You must do this in a loop, calling `read()` repeatedly until it returns `null`. It is necessary to completely drain the stream's buffer like this in order to trigger a new "readable" event in the future. If you stop calling `read()` while there is still readable data, you will not get another "readable" event and your program is likely to hang.

Streams in paused mode emit "end" and "error" events just like flowing mode streams do. If you are writing a program that reads data from a Readable stream and writes it to a Writable stream, then paused mode may not be a good choice. In order to properly handle backpressure, you only want to read when the input stream is readable and the output stream is not backed up. In paused mode, that means reading and writing until `read()` returns `null` or `write()` returns `false`, and then starting reading or writing again on a `readable` or `drain` event. This is inelegant, and you may find that flowing mode (or pipes) is easier in this case.

The following code demonstrates how you can compute a SHA256 hash for the contents of a specified file. It uses a Readable stream in paused mode to read the contents of a file in chunks, then passes each chunk to the object that computes the hash. (Note that in Node 12 and later, it would be simpler to write this function using a `for/await` loop.)

```
const fs = require("fs");
const crypto = require("crypto");

// Compute a sha256 hash of the contents of the named file and pass the
// hash (as a string) to the specified error-first callback function.
function sha256(filename, callback) {
    let input = fs.createReadStream(filename); // The data stream.
    let hasher = crypto.createHash("sha256");  // For computing the hash.

    input.on("readable", () => {         // When there is data ready to read
        let chunk;
        while(chunk = input.read()) {    // Read a chunk, and if non-null,
            hasher.update(chunk);        // pass it to the hasher,
        }                                // and keep looping until not readable
    });
    input.on("end", () => {              // At the end of the stream,
        let hash = hasher.digest("hex"); // compute the hash,
        callback(null, hash);            // and pass it to the callback.
    });
    input.on("error", callback);         // On error, call callback
}

// Here's a simple command-line utility to compute the hash of a file
sha256(process.argv[2], (err, hash) => { // Pass filename from command line.
    if (err) {                           // If we get an error
        console.error(err.toString());   // print it as an error.
    } else {                             // Otherwise,
        console.log(hash);               // print the hash string.
    }
});
```

16.6 Process, CPU, and Operating System Details

The global Process object has a number of useful properties and functions that generally relate to the state of the currently running Node process. Consult the Node documentation for complete details, but here are some properties and functions you should be aware of:

```
process.argv            // An array of command-line arguments.
process.arch            // The CPU architecture: "x64", for example.
process.cwd()           // Returns the current working directory.
process.chdir()         // Sets the current working directory.
process.cpuUsage()      // Reports CPU usage.
process.env             // An object of environment variables.
process.execPath        // The absolute filesystem path to the node executable.
process.exit()          // Terminates the program.
process.exitCode        // An integer code to be reported when the program exits.
process.getuid()        // Return the Unix user id of the current user.
process.hrtime.bigint() // Return a "high-resolution" nanosecond timestamp.
process.kill()          // Send a signal to another process.
process.memoryUsage()   // Return an object with memory usage details.
```

```
process.nextTick()      // Like setImmediate(), invoke a function soon.
process.pid             // The process id of the current process.
process.ppid            // The parent process id.
process.platform        // The OS: "linux", "darwin", or "win32", for example.
process.resourceUsage() // Return an object with resource usage details.
process.setuid()        // Sets the current user, by id or name.
process.title           // The process name that appears in `ps` listings.
process.umask()         // Set or return the default permissions for new files.
process.uptime()        // Return Node's uptime in seconds.
process.version         // Node's version string.
process.versions        // Version strings for the libraries Node depends on.
```

The "os" module (which, unlike `process`, needs to be explicitly loaded with `require()`) provides access to similarly low-level details about the computer and operating system that Node is running on. You may never need to use any of these features, but it is worth knowing that Node makes them available:

```
const os = require("os");
os.arch()              // Returns CPU architecture. "x64" or "arm", for example.
os.constants           // Useful constants such as os.constants.signals.SIGINT.
os.cpus()              // Data about system CPU cores, including usage times.
os.endianness()        // The CPU's native endianness "BE" or "LE".
os.EOL                 // The OS native line terminator: "\n" or "\r\n".
os.freemem()           // Returns the amount of free RAM in bytes.
os.getPriority()       // Returns the OS scheduling priority of a process.
os.homedir()           // Returns the current user's home directory.
os.hostname()          // Returns the hostname of the computer.
os.loadavg()           // Returns the 1, 5, and 15-minute load averages.
os.networkInterfaces() // Returns details about available network. connections.
os.platform()          // Returns OS: "linux", "darwin", or "win32", for example.
os.release()           // Returns the version number of the OS.
os.setPriority()       // Attempts to set the scheduling priority for a process.
os.tmpdir()            // Returns the default temporary directory.
os.totalmem()          // Returns the total amount of RAM in bytes.
os.type()              // Returns OS: "Linux", "Darwin", or "Windows_NT", e.g.
os.uptime()            // Returns the system uptime in seconds.
os.userInfo()          // Returns uid, username, home, and shell of current user.
```

16.7 Working with Files

Node's "fs" module is a comprehensive API for working with files and directories. It is complemented by the "path" module, which defines utility functions for working with file and directory names. The "fs" module contains a handful of high-level functions for easily reading, writing, and copying files. But most of the functions in the module are low-level JavaScript bindings to Unix system calls (and their equivalents on Windows). If you have worked with low-level filesystem calls before (in C or other languages), then the Node API will be familiar to you. If not, you may find parts of the "fs" API to be terse and unintuitive. The function to delete a file, for example, is called `unlink()`.

The "fs" module defines a large API, mainly because there are usually multiple variants of each fundamental operation. As discussed at the beginning of the chapter, most functions such as `fs.readFile()` are nonblocking, callback-based, and asynchronous. Typically, though, each of these functions has a synchronous blocking variant, such as `fs.readFileSync()`. In Node 10 and later, many of these functions also have a Promise-based asynchronous variant such as `fs.promises.readFile()`. Most "fs" functions take a string as their first argument, specifying the path (filename plus optional directory names) to the file that is to be operated on. But a number of these functions also support a variant that takes an integer "file descriptor" as the first argument instead of a path. These variants have names that begin with the letter "f." For example, `fs.truncate()` truncates a file specified by path, and `fs.ftruncate()` truncates a file specified by file descriptor. There is a Promise-based `fs.promises.truncate()` that expects a path and another Promise-based version that is implemented as a method of a FileHandle object. (The FileHandle class is the equivalent of a file descriptor in the Promise-based API.) Finally, there are a handful of functions in the "fs" module that have variants whose names are prefixed with the letter "l." These "l" variants are like the base function but do not follow symbolic links in the filesystem and instead operate directly on the symbolic links themselves.

16.7.1 Paths, File Descriptors, and FileHandles

In order to use the "fs" module to work with files, you first need to be able to name the file you want to work with. Files are most often specified by *path*, which means the name of the file itself, plus the hierarchy of directories in which the file appears. If a path is *absolute*, it means that directories all the way up to the filesystem root are specified. Otherwise, the path is *relative* and is only meaningful in relation to some other path, usually the *current working directory*. Working with paths can be a little tricky because different operating systems use different characters to separate directory names, it is easy to accidentally double those separator characters when concatenating paths, and because `../` parent directory path segments need special handling. Node's "path" module and a couple of other important Node features help:

```
// Some important paths
process.cwd()      // Absolute path of the current working directory.
__filename         // Absolute path of the file that holds the current code.
__dirname          // Absolute path of the directory that holds __filename.
os.homedir()       // The user's home directory.

const path = require("path");

path.sep                        // Either "/" or "\" depending on your OS

// The path module has simple parsing functions
let p = "src/pkg/test.js";      // An example path
path.basename(p)                // => "test.js"
path.extname(p)                 // => ".js"
```

```
path.dirname(p)                 // => "src/pkg"
path.basename(path.dirname(p))  // => "pkg"
path.dirname(path.dirname(p))   // => "src"

// normalize() cleans up paths:
path.normalize("a/b/c/../d/")   // => "a/b/d/": handles ../ segments
path.normalize("a/./b")         // => "a/b": strips "./" segments
path.normalize("//a//b//")      // => "/a/b/": removes duplicate /

// join() combines path segments, adding separators, then normalizes
path.join("src", "pkg", "t.js")  // => "src/pkg/t.js"

// resolve() takes one or more path segments and returns an absolute
// path. It starts with the last argument and works backward, stopping
// when it has built an absolute path or resolving against process.cwd().
path.resolve()                  // => process.cwd()
path.resolve("t.js")            // => path.join(process.cwd(), "t.js")
path.resolve("/tmp", "t.js")    // => "/tmp/t.js"
path.resolve("/a", "/b", "t.js") // => "/b/t.js"
```

Note that `path.normalize()` is simply a string manipulation function that has no access to the actual filesystem. The `fs.realpath()` and `fs.realpathSync()` functions perform filesystem-aware canonicalization: they resolve symbolic links and interpret relative pathnames relative to the current working directory.

In the previous examples, we assumed that the code is running on a Unix-based OS and `path.sep` is "/." If you want to work with Unix-style paths even when on a Windows system, then use `path.posix` instead of `path`. And conversely, if you want to work with Windows paths even when on a Unix system, `path.win32`. `path.posix` and `path.win32` define the same properties and functions as `path` itself.

Some of the "fs" functions we'll be covering in the next sections expect a *file descriptor* instead of a file name. File descriptors are integers used as OS-level references to "open" files. You obtain a descriptor for a given name by calling the `fs.open()` (or `fs.openSync()`) function. Processes are only allowed to have a limited number of files open at one time, so it is important that you call `fs.close()` on your file descriptors when you are done with them. You need to open files if you want to use the lowest-level `fs.read()` and `fs.write()` functions that allow you to jump around within a file, reading and writing bits of it at different times. There are other functions in the "fs" module that use file descriptors, but they all have name-based versions, and it only really makes sense to use the descriptor-based functions if you were going to open the file to read or write anyway.

Finally, in the Promise-based API defined by `fs.promises`, the equivalent of `fs.open()` is `fs.promises.open()`, which returns a Promise that resolves to a FileHandle object. This FileHandle object serves the same purpose as a file descriptor. Again, however, unless you need to use the lowest-level `read()` and `write()` methods

of a FileHandle, there is really no reason to create one. And if you do create a FileHandle, you should remember to call its `close()` method once you are done with it.

16.7.2 Reading Files

Node allows you to read file content all at once, via a stream, or with the low-level API.

If your files are small, or if memory usage and performance are not the highest priority, then it is often easiest to read the entire content of a file with a single call. You can do this synchronously, with a callback, or with a Promise. By default, you'll get the bytes of the file as a buffer, but if you specify an encoding, you'll get a decoded string instead.

```
const fs = require("fs");
let buffer = fs.readFileSync("test.data");      // Synchronous, returns buffer
let text = fs.readFileSync("data.csv", "utf8"); // Synchronous, returns string

// Read the bytes of the file asynchronously
fs.readFile("test.data", (err, buffer) => {
    if (err) {
        // Handle the error here
    } else {
        // The bytes of the file are in buffer
    }
});

// Promise-based asynchronous read
fs.promises
    .readFile("data.csv", "utf8")
    .then(processFileText)
    .catch(handleReadError);

// Or use the Promise API with await inside an async function
async function processText(filename, encoding="utf8") {
    let text = await fs.promises.readFile(filename, encoding);
    // ... process the text here...
}
```

If you are able to process the contents of a file sequentially and do not need to have the entire content of the file in memory at the same time, then reading a file via a stream may be the most efficient approach. We've covered streams extensively: here is how you might use a stream and the `pipe()` method to write the contents of a file to standard output:

```
function printFile(filename, encoding="utf8") {
    fs.createReadStream(filename, encoding).pipe(process.stdout);
}
```

Finally, if you need low-level control over exactly what bytes you read from a file and when you read them, you can open a file to get a file descriptor and then use `fs.read()`, `fs.readSync()`, or `fs.promises.read()` to read a specified number of bytes from a specified source location of the file into a specified buffer at the specified destination position:

```
const fs = require("fs");

// Reading a specific portion of a data file
fs.open("data", (err, fd) => {
    if (err) {
        // Report error somehow
        return;
    }
    try {
        // Read bytes 20 through 420 into a newly allocated buffer.
        fs.read(fd, Buffer.alloc(400), 0, 400, 20, (err, n, b) => {
            // err is the error, if any.
            // n is the number of bytes actually read
            // b is the buffer that they bytes were read into.
        });
    }
    finally {          // Use a finally clause so we always
        fs.close(fd);  // close the open file descriptor
    }
});
```

The callback-based `read()` API is awkward to use if you need to read more than one chunk of data from a file. If you can use the synchronous API (or the Promise-based API with `await`), it becomes easy to read multiple chunks from a file:

```
const fs = require("fs");

function readData(filename) {
    let fd = fs.openSync(filename);
    try {
        // Read the file header
        let header = Buffer.alloc(12); // A 12 byte buffer
        fs.readSync(fd, header, 0, 12, 0);

        // Verify the file's magic number
        let magic = header.readInt32LE(0);
        if (magic !== 0xDADAFEED) {
            throw new Error("File is of wrong type");
        }

        // Now get the offset and length of the data from the header
        let offset = header.readInt32LE(4);
        let length = header.readInt32LE(8);

        // And read those bytes from the file
        let data = Buffer.alloc(length);
```

```
            fs.readSync(fd, data, 0, length, offset);
            return data;
        } finally {
            // Always close the file, even if an exception is thrown above
            fs.closeSync(fd);
        }
    }
```

16.7.3 Writing Files

Writing files in Node is a lot like reading them, with a few extra details that you need to know about. One of these details is that the way you create a new file is simply by writing to a filename that does not already exist.

As with reading, there are three basic ways to write files in Node. If you have the entire content of the file in a string or a buffer, you can write the entire thing in one call with `fs.writeFile()` (callback-based), `fs.writeFileSync()` (synchronous), or `fs.promises.writeFile()` (Promise-based):

```
fs.writeFileSync(path.resolve(__dirname, "settings.json"),
                 JSON.stringify(settings));
```

If the data you are writing to the file is a string, and you want to use an encoding other than "utf8," pass the encoding as an optional third argument.

The related functions `fs.appendFile()`, `fs.appendFileSync()`, and `fs.promises.appendFile()` are similar, but when the specified file already exists, they append their data to the end rather than overwriting the existing file content.

If the data you want to write to a file is not all in one chunk, or if it is not all in memory at the same time, then using a Writable stream is a good approach, assuming that you plan to write the data from beginning to end without skipping around in the file:

```
const fs = require("fs");
let output = fs.createWriteStream("numbers.txt");
for(let i = 0; i < 100; i++) {
    output.write(`${i}\n`);
}
output.end();
```

Finally, if you want to write data to a file in multiple chunks, and you want to be able to control the exact position within the file at which each chunk is written, then you can open the file with `fs.open()`, `fs.openSync()`, or `fs.promises.open()` and then use the resulting file descriptor with the `fs.write()` or `fs.writeSync()` functions. These functions come in different forms for strings and buffers. The string variant takes a file descriptor, a string, and the file position at which to write that string (with an encoding as an optional fourth argument). The buffer variant takes a file descriptor, a buffer, an offset, and a length that specify a chunk of data within the buffer, and a file position at which to write the bytes of that chunk. And if you have an array of

Buffer objects that you want to write, you can do this with a single `fs.writev()` or `fs.writevSync()`. Similar low-level functions exist for writing buffers and strings using `fs.promises.open()` and the FileHandle object it produces.

File Mode Strings

We saw the `fs.open()` and `fs.openSync()` methods before when using the low-level API to read files. In that use case, it was sufficient to just pass the filename to the open function. When you want to write a file, however, you must also specify a second string argument that specifies how you intend to use the file descriptor. Some of the available flag strings are as follows:

`"w"`
: Open the file for writing

`"w+"`
: Open for writing and reading

`"wx"`
: Open for creating a new file; fails if the named file already exists

`"wx+"`
: Open for creation, and also allow reading; fails if the named file already exists

`"a"`
: Open the file for appending; existing content won't be overwritten

`"a+"`
: Open for appending, but also allow reading

If you do not pass one of these flag strings to `fs.open()` or `fs.openSync()`, they use the default "r" flag, making the file descriptor read-only. Note that it can also be useful to pass these flags to other file-writing methods:

```
// Write to a file in one call, but append to anything that is already there.
// This works like fs.appendFileSync()
fs.writeFileSync("messages.log", "hello", { flag: "a" });

// Open a write stream, but throw an error if the file already exists.
// We don't want to accidentally overwrite something!
// Note that the option above is "flag" and is "flags" here
fs.createWriteStream("messages.log", { flags: "wx" });
```

You can chop off the end of a file with `fs.truncate()`, `fs.truncateSync()`, or `fs.promises.truncate()`. These functions take a path as their first argument and a length as their second, and modify the file so that it has the specified length. If you omit the length, zero is used and the file becomes empty. Despite the name of these

functions, they can also be used to extend a file: if you specify a length that is longer than the current file size, the file is extended with zero bytes to the new size. If you have already opened the file you wish to modify, you can use `ftruncate()` or `ftruncateSync()` with the file descriptor or FileHandle.

The various file-writing functions described here return or invoke their callback or resolve their Promise when the data has been "written" in the sense that Node has handed it off to the operating system. But this does not necessarily mean that the data has actually been written to persistent storage yet: at least some of your data may still be buffered somewhere in the operating system or in a device driver waiting to be written to disk. If you call `fs.writeSync()` to synchronously write some data to a file, and if there is a power outage immediately after the function returns, you may still lose data. If you want to force your data out to disk so you know for sure that it has been safely saved, use `fs.fsync()` or `fs.fsyncSync()`. These functions only work with file descriptors: there is no path-based version.

16.7.4 File Operations

The preceding discussion of Node's stream classes included two examples of `copyFile()` functions. These are not practical utilities that you would actually use because the "fs" module defines its own `fs.copyFile()` method (and also `fs.copyFileSync()` and `fs.promises.copyFile()`, of course).

These functions take the name of the original file and the name of the copy as their first two arguments. These can be specified as strings or as URL or Buffer objects. An optional third argument is an integer whose bits specify flags that control details of the copy operation. And for the callback-based `fs.copyFile()`, the final argument is a callback function that will be called with no arguments when the copy is complete, or that will be called with an error argument if something fails. Following are some examples:

```
// Basic synchronous file copy.
fs.copyFileSync("ch15.txt", "ch15.bak");

// The COPYFILE_EXCL argument copies only if the new file does not already
// exist. It prevents copies from overwriting existing files.
fs.copyFile("ch15.txt", "ch16.txt", fs.constants.COPYFILE_EXCL, err => {
    // This callback will be called when done. On error, err will be non-null.
});

// This code demonstrates the Promise-based version of the copyFile function.
// Two flags are combined with the bitwise OR opeartor |. The flags mean that
// existing files won't be overwritten, and that if the filesystem supports
// it, the copy will be a copy-on-write clone of the original file, meaning
// that no additional storage space will be required until either the original
// or the copy is modified.
fs.promises.copyFile("Important data",
```

```
                    `Important data ${new Date().toISOString()}"
                    fs.constants.COPYFILE_EXCL | fs.constants.COPYFILE_FICLONE)
    .then(() => {
        console.log("Backup complete");
    });
    .catch(err => {
        console.error("Backup failed", err);
    });
```

The `fs.rename()` function (along with the usual synchronous and Promise-based variants) moves and/or renames a file. Call it with the current path to the file and the desired new path to the file. There is no flags argument, but the callback-based version takes a callback as the third argument:

```
fs.renameSync("ch15.bak", "backups/ch15.bak");
```

Note that there is no flag to prevent renaming from overwriting an existing file. Also keep in mind that files can only be renamed within a filesystem.

The functions `fs.link()` and `fs.symlink()` and their variants have the same signatures as `fs.rename()` and behave something like `fs.copyFile()` except that they create hard links and symbolic links, respectively, rather than creating a copy.

Finally, `fs.unlink()`, `fs.unlinkSync()`, and `fs.promises.unlink()` are Node's functions for deleting a file. (The unintuitive naming is inherited from Unix where deleting a file is basically the opposite of creating a hard link to it.) Call this function with the string, buffer, or URL path to the file to be deleted, and pass a callback if you are using the callback-based version:

```
fs.unlinkSync("backups/ch15.bak");
```

16.7.5 File Metadata

The `fs.stat()`, `fs.statSync()`, and `fs.promises.stat()` functions allow you to obtain metadata for a specified file or directory. For example:

```
const fs = require("fs");
let stats = fs.statSync("book/ch15.md");
stats.isFile()         // => true: this is an ordinary file
stats.isDirectory()    // => false: it is not a directory
stats.size             // file size in bytes
stats.atime            // access time: Date when it was last read
stats.mtime            // modification time: Date when it was last written
stats.uid              // the user id of the file's owner
stats.gid              // the group id of the file's owner
stats.mode.toString(8) // the file's permissions, as an octal string
```

The returned Stats object contains other, more obscure properties and methods, but this code demonstrates those that you are most likely to use.

`fs.lstat()` and its variants work just like `fs.stat()`, except that if the specified file is a symbolic link, Node will return metadata for the link itself rather than following the link.

If you have opened a file to produce a file descriptor or a FileHandle object, then you can use `fs.fstat()` or its variants to get metadata information for the opened file without having to specify the filename again.

In addition to querying metadata with `fs.stat()` and all of its variants, there are also functions for changing metadata.

`fs.chmod()`, `fs.lchmod()`, and `fs.fchmod()` (along with synchronous and Promise-based versions) set the "mode" or permissions of a file or directory. Mode values are integers in which each bit has a specific meaning and are easiest to think about in octal notation. For example, to make a file read-only to its owner and inaccessible to everyone else, use `0o400`:

```
fs.chmodSync("ch15.md", 0o400);  // Don't delete it accidentally!
```

`fs.chown()`, `fs.lchown()`, and `fs.fchown()` (along with synchronous and Promise-based versions) set the owner and group (as IDs) for a file or directory. (These matter because they interact with the file permissions set by `fs.chmod()`.)

Finally, you can set the access time and modification time of a file or directory with `fs.utimes()` and `fs.futimes()` and their variants.

16.7.6 Working with Directories

To create a new directory in Node, use `fs.mkdir()`, `fs.mkdirSync()`, or `fs.promises.mkdir()`. The first argument is the path of the directory to be created. The optional second argument can be an integer that specifies the mode (permissions bits) for the new directory. Or you can pass an object with optional `mode` and `recursive` properties. If `recursive` is `true`, then this function will create any directories in the path that do not already exist:

```
// Ensure that dist/ and dist/lib/ both exist.
fs.mkdirSync("dist/lib", { recursive: true });
```

`fs.mkdtemp()` and its variants take a path prefix you provide, append some random characters to it (this is important for security), create a directory with that name, and return (or pass to a callback) the directory path to you.

To delete a directory, use `fs.rmdir()` or one of its variants. Note that directories must be empty before they can be deleted:

```
// Create a random temporary directory and get its path, then
// delete it when we are done
let tempDirPath;
try {
```

```
    tempDirPath = fs.mkdtempSync(path.join(os.tmpdir(), "d"));
    // Do something with the directory here
} finally {
    // Delete the temporary directory when we're done with it
    fs.rmdirSync(tempDirPath);
}
```

The "fs" module provides two distinct APIs for listing the contents of a directory. First, `fs.readdir()`, `fs.readdirSync()`, and `fs.promises.readdir()` read the entire directory all at once and give you an array of strings or an array of Dirent objects that specify the names and types (file or directory) of each item. Filenames returned by these functions are just the local name of the file, not the entire path. Here are examples:

```
let tempFiles = fs.readdirSync("/tmp");  // returns an array of strings

// Use the Promise-based API to get a Dirent array, and then
// print the paths of subdirectories
fs.promises.readdir("/tmp", {withFileTypes: true})
    .then(entries => {
        entries.filter(entry => entry.isDirectory())
            .map(entry => entry.name)
            .forEach(name => console.log(path.join("/tmp/", name)));
    })
    .catch(console.error);
```

If you anticipate needing to list directories that might have thousands of entries, you might prefer the streaming approach of `fs.opendir()` and its variants. These functions return a Dir object representing the specified directory. You can use the `read()` or `readSync()` methods of the Dir object to read one Dirent at a time. If you pass a callback function to `read()`, it will call the callback. And if you omit the callback argument, it will return a Promise. When there are no more directory entries, you'll get `null` instead of a Dirent object.

The easiest way to use Dir objects is as async iterators with a `for/await` loop. Here, for example, is a function that uses the streaming API to list directory entries, calls `stat()` on each entry, and prints file and directory names and sizes:

```
const fs = require("fs");
const path = require("path");

async function listDirectory(dirpath) {
    let dir = await fs.promises.opendir(dirpath);
    for await (let entry of dir) {
        let name = entry.name;
        if (entry.isDirectory()) {
            name += "/";  // Add a trailing slash to subdirectories
        }
        let stats = await fs.promises.stat(path.join(dirpath, name));
        let size = stats.size;
```

```
            console.log(String(size).padStart(10), name);
        }
    }
```

16.8 HTTP Clients and Servers

Node's "http," "https," and "http2" modules are full-featured but relatively low-level implementations of the HTTP protocols. They define comprehensive APIs for implementing HTTP clients and servers. Because the APIs are relatively low-level, there is not room in this chapter to cover all the features. But the examples that follow demonstrate how to write basic clients and servers.

The simplest way to make a basic HTTP GET request is with `http.get()` or `https.get()`. The first argument to these functions is the URL to fetch. (If it is an `http://` URL, you must use the "http" module, and if it is an `https://` URL you must use the "https" module.) The second argument is a callback that will be invoked with an IncomingMessage object when the server's response has started to arrive. When the callback is called, the HTTP status and headers are available, but the body may not be ready yet. The IncomingMessage object is a Readable stream, and you can use the techniques demonstrated earlier in this chapter to read the response body from it.

The `getJSON()` function at the end of §13.2.6 used the `http.get()` function as part of a demonstration of the `Promise()` constructor. Now that you know about Node streams and the Node programming model more generally, it is worth revisiting that example to see how `http.get()` is used.

`http.get()` and `https.get()` are slightly simplified variants of the more general `http.request()` and `https.request()` functions. The following `postJSON()` function demonstrates how to use `https.request()` to make an HTTPS POST request that includes a JSON request body. Like the `getJSON()` function of Chapter 13, it expects a JSON response and returns a Promise that fulfills to the parsed version of that response:

```
const https = require("https");

/*
 * Convert the body object to a JSON string then HTTPS POST it to the
 * specified API endpoint on the specified host. When the response arrives,
 * parse the response body as JSON and resolve the returned Promise with
 * that parsed value.
 */
function postJSON(host, endpoint, body, port, username, password) {
    // Return a Promise object immediately, then call resolve or reject
    // when the HTTPS request succeeds or fails.
    return new Promise((resolve, reject) => {
        // Convert the body object to a string
```

```
    let bodyText = JSON.stringify(body);

    // Configure the HTTPS request
    let requestOptions = {
        method: "POST",       // Or "GET", "PUT", "DELETE", etc.
        host: host,           // The host to connect to
        path: endpoint,       // The URL path
        headers: {            // HTTP headers for the request
            "Content-Type": "application/json",
            "Content-Length": Buffer.byteLength(bodyText)
        }
    };

    if (port) {                     // If a port is specified,
        requestOptions.port = port;  // use it for the request.
    }
    // If credentials are specified, add an Authorization header.
    if (username && password) {
        requestOptions.auth = `${username}:${password}`;
    }

    // Now create the request based on the configuration object
    let request = https.request(requestOptions);

    // Write the body of the POST request and end the request.
    request.write(bodyText);
    request.end();

    // Fail on request errors (such as no network connection)
    request.on("error", e => reject(e));

    // Handle the response when it starts to arrive.
    request.on("response", response => {
        if (response.statusCode !== 200) {
            reject(new Error(`HTTP status ${response.statusCode}`));
            // We don't care about the response body in this case, but
            // we don't want it to stick around in a buffer somewhere, so
            // we put the stream into flowing mode without registering
            // a "data" handler so that the body is discarded.
            response.resume();
            return;
        }

        // We want text, not bytes. We're assuming the text will be
        // JSON-formatted but aren't bothering to check the
        // Content-Type header.
        response.setEncoding("utf8");

        // Node doesn't have a streaming JSON parser, so we read the
        // entire response body into a string.
        let body = "";
        response.on("data", chunk => { body += chunk; });
```

```
            // And now handle the response when it is complete.
            response.on("end", () => {          // When the response is done,
                try {                           // try to parse it as JSON
                    resolve(JSON.parse(body));  // and resolve the result.
                } catch(e) {                    // Or, if anything goes wrong,
                    reject(e);                  // reject with the error
                }
            });
        });
    });
}
```

In addition to making HTTP and HTTPS requests, the "http" and "https" modules also allow you to write servers that respond to those requests. The basic approach is as follows:

- Create a new Server object.
- Call its `listen()` method to begin listening for requests on a specified port.
- Register an event handler for "request" events, use that handler to read the client's request (particularly the `request.url` property), and write your response.

The code that follows creates a simple HTTP server that serves static files from the local filesystem and also implements a debugging endpoint that responds to a client's request by echoing that request.

```
// This is a simple static HTTP server that serves files from a specified
// directory. It also implements a special /test/mirror endpoint that
// echoes the incoming request, which can be useful when debugging clients.
const http = require("http");   // Use "https" if you have a certificate
const url = require("url");     // For parsing URLs
const path = require("path");   // For manipulating filesystem paths
const fs = require("fs");       // For reading files

// Serve files from the specified root directory via an HTTP server that
// listens on the specified port.
function serve(rootDirectory, port) {
    let server = new http.Server();  // Create a new HTTP server
    server.listen(port);             // Listen on the specified port
    console.log("Listening on port", port);

    // When requests come in, handle them with this function
    server.on("request", (request, response) => {
        // Get the path portion of the request URL, ignoring
        // any query parameters that are appended to it.
        let endpoint = url.parse(request.url).pathname;

        // If the request was for "/test/mirror", send back the request
        // verbatim. Useful when you need to see the request headers and body.
        if (endpoint === "/test/mirror") {
```

```
        // Set response header
        response.setHeader("Content-Type", "text/plain; charset=UTF-8");

        // Specify response status code
        response.writeHead(200);  // 200 OK

        // Begin the response body with the request
        response.write(`${request.method} ${request.url} HTTP/${
                          request.httpVersion
                      }\r\n`);

        // Output the request headers
        let headers = request.rawHeaders;
        for(let i = 0; i < headers.length; i += 2) {
            response.write(`${headers[i]}: ${headers[i+1]}\r\n`);
        }

        // End headers with an extra blank line
        response.write("\r\n");

        // Now we need to copy any request body to the response body
        // Since they are both streams, we can use a pipe
        request.pipe(response);
    }
    // Otherwise, serve a file from the local directory.
    else {
        // Map the endpoint to a file in the local filesystem
        let filename = endpoint.substring(1); // strip leading /
        // Don't allow "../" in the path because it would be a security
        // hole to serve anything outside the root directory.
        filename = filename.replace(/\.\.\//g, "");
        // Now convert from relative to absolute filename
        filename = path.resolve(rootDirectory, filename);

        // Now guess the type file's content type based on extension
        let type;
        switch(path.extname(filename))  {
        case ".html":
        case ".htm": type = "text/html"; break;
        case ".js":  type = "text/javascript"; break;
        case ".css": type = "text/css"; break;
        case ".png": type = "image/png"; break;
        case ".txt": type = "text/plain"; break;
        default:     type = "application/octet-stream"; break;
        }

        let stream = fs.createReadStream(filename);
        stream.once("readable", () => {
            // If the stream becomes readable, then set the
            // Content-Type header and a 200 OK status. Then pipe the
            // file reader stream to the response. The pipe will
            // automatically call response.end() when the stream ends.
```

```
                response.setHeader("Content-Type", type);
                response.writeHead(200);
                stream.pipe(response);
            });

            stream.on("error", (err) => {
                // Instead, if we get an error trying to open the stream
                // then the file probably does not exist or is not readable.
                // Send a 404 Not Found plain-text response with the
                // error message.
                response.setHeader("Content-Type", "text/plain; charset=UTF-8");
                response.writeHead(404);
                response.end(err.message);
            });
        }
    });
}

// When we're invoked from the command line, call the serve() function
serve(process.argv[2] || "/tmp", parseInt(process.argv[3]) || 8000);
```

Node's built-in modules are all you need to write simple HTTP and HTTPS servers. Note, however, that production servers are not typically built directly on top of these modules. Instead, most nontrivial servers are implemented using external libraries—such as the Express framework—that provide "middleware" and other higher-level utilities that backend web developers have come to expect.

16.9 Non-HTTP Network Servers and Clients

Web servers and clients have become so ubiquitous that it is easy to forget that it is possible to write clients and servers that do not use HTTP. Even though Node has a reputation as a good environment for writing web servers, Node also has full support for writing other types of network servers and clients.

If you are comfortable working with streams, then networking is relatively simple, because network sockets are simply a kind of Duplex stream. The "net" module defines Server and Socket classes. To create a server, call `net.createServer()`, then call the `listen()` method of the resulting object to tell the server what port to listen on for connections. The Server object will generate "connection" events when a client connects on that port, and the value passed to the event listener will be a Socket object. The Socket object is a Duplex stream, and you can use it to read data from the client and write data to the client. Call `end()` on the Socket to disconnect.

Writing a client is even easier: pass a port number and hostname to `net.createConnection()` to create a socket to communicate with whatever server is running on that host and listening on that port. Then use that socket to read and write data from and to the server.

The following code demonstrates how to write a server with the "net" module. When the client connects, the server tells a knock-knock joke:

```
// A TCP server that delivers interactive knock-knock jokes on port 6789.
// (Why is six afraid of seven? Because seven ate nine!)
const net = require("net");
const readline = require("readline");

// Create a Server object and start listening for connections
let server = net.createServer();
server.listen(6789, () => console.log("Delivering laughs on port 6789"));

// When a client connects, tell them a knock-knock joke.
server.on("connection", socket => {
    tellJoke(socket)
        .then(() => socket.end())  // When the joke is done, close the socket.
        .catch((err) => {
            console.error(err);    // Log any errors that occur,
            socket.end();          // but still close the socket!
        });
});

// These are all the jokes we know.
const jokes = {
    "Boo": "Don't cry...it's only a joke!",
    "Lettuce": "Let us in! It's freezing out here!",
    "A little old lady": "Wow, I didn't know you could yodel!"
};

// Interactively perform a knock-knock joke over this socket, without blocking.
async function tellJoke(socket) {
    // Pick one of the jokes at random
    let randomElement = a => a[Math.floor(Math.random() * a.length)];
    let who = randomElement(Object.keys(jokes));
    let punchline = jokes[who];

    // Use the readline module to read the user's input one line at a time.
    let lineReader = readline.createInterface({
        input: socket,
        output: socket,
        prompt: ">> "
    });

    // A utility function to output a line of text to the client
    // and then (by default) display a prompt.
    function output(text, prompt=true) {
        socket.write(`${text}\r\n`);
        if (prompt) lineReader.prompt();
    }

    // Knock-knock jokes have a call-and-response structure.
    // We expect different input from the user at different stages and
```

```
        // take different action when we get that input at different stages.
        let stage = 0;

        // Start the knock-knock joke off in the traditional way.
        output("Knock knock!");

        // Now read lines asynchronously from the client until the joke is done.
        for await (let inputLine of lineReader) {
            if (stage === 0) {
                if (inputLine.toLowerCase() === "who's there?") {
                    // If the user gives the right response at stage 0
                    // then tell the first part of the joke and go to stage 1.
                    output(who);
                    stage = 1;
                } else  {
                    // Otherwise teach the user how to do knock-knock jokes.
                    output('Please type "Who\'s there?".');
                }
            } else if (stage === 1) {
                if (inputLine.toLowerCase() === `${who.toLowerCase()} who?`) {
                    // If the user's response is correct at stage 1, then
                    // deliver the punchline and return since the joke is done.
                    output(`${punchline}`, false);
                    return;
                } else {
                    // Make the user play along.
                    output(`Please type "${who} who?".`);
                }
            }
        }
    }
```

Simple text-based servers like this do not typically need a custom client. If the `nc` ("netcat") utility is installed on your system, you can use it to communicate with this server as follows:

```
$ nc localhost 6789
Knock knock!
>> Who's there?
A little old lady
>> A little old lady who?
Wow, I didn't know you could yodel!
```

On the other hand, writing a custom client for the joke server is easy in Node. We just connect to the server, then pipe the server's output to stdout and pipe stdin to the server's input:

```
// Connect to the joke port (6789) on the server named on the command line
let socket = require("net").createConnection(6789, process.argv[2]);
socket.pipe(process.stdout);              // Pipe data from the socket to stdout
process.stdin.pipe(socket);               // Pipe data from stdin to the socket
socket.on("close", () => process.exit()); // Quit when the socket closes.
```

In addition to supporting TCP-based servers, Node's "net" module also supports interprocess communication over "Unix domain sockets" that are identified by a file-system path rather than by a port number. We are not going to cover that kind of socket in this chapter, but the Node documentation has details. Other Node features that we don't have space to cover here include the "dgram" module for UDP-based clients and servers and the "tls" module that is to "net" as "https" is to "http." The `tls.Server` and `tls.TLSSocket` classes allow the creation of TCP servers (like the knock-knock joke server) that use SSL-encrypted connections like HTTPS servers do.

16.10 Working with Child Processes

In addition to writing highly concurrent servers, Node also works well for writing scripts that execute other programs. In Node the "child_process" module defines a number of functions for running other programs as child processes. This section demonstrates some of those functions, starting with the simplest and moving to the more complicated.

16.10.1 execSync() and execFileSync()

The easiest way to run another program is with `child_process.execSync()`. This function takes the command to run as its first argument. It creates a child process, runs a shell in that process, and uses the shell to execute the command you passed. Then it blocks until the command (and the shell) exit. If the command exits with an error, then `execSync()` throws an exception. Otherwise, `execSync()` returns whatever output the command writes to its stdout stream. By default this return value is a buffer, but you can specify an encoding in an optional second argument to get a string instead. If the command writes any output to stderr, that output just gets passed through to the parent process's stderr stream.

So, for example, if you are writing a script and performance is not a concern, you might use `child_process.execSync()` to list a directory with a familiar Unix shell command rather than using the `fs.readdirSync()` function:

```
const child_process = require("child_process");
let listing = child_process.execSync("ls -l web/*.html", {encoding: "utf8"});
```

The fact that `execSync()` invokes a full Unix shell means that the string you pass to it can include multiple semicolon-separated commands, and can take advantage of shell features such as filename wildcards, pipes, and output redirection. This also means that you must be careful to never pass a command to `execSync()` if any portion of that command is user input or comes from a similar untrusted source. The complex syntax of shell commands can be easily subverted to allow an attacker to run arbitrary code.

If you don't need the features of a shell, you can avoid the overhead of starting a shell by using `child_process.execFileSync()`. This function executes a program directly, without invoking a shell. But since no shell is involved, it can't parse a command line, and you must pass the executable as the first argument and an array of command-line arguments as the second argument:

```
let listing = child_process.execFileSync("ls", ["-l", "web/"],
                                         {encoding: "utf8"});
```

Child Process Options

`execSync()` and many of the other `child_process` functions have a second or third optional argument that specifies additional details about how the child process is to run. The `encoding` property of this object was used earlier to specify that we'd like the command output to be delivered as a string rather than as a buffer. Other important properties that you can specify include the following (note that not all options are available to all child process functions):

- `cwd` specifies the working directory for the child process. If you omit this, then the child process inherits the value of `process.cwd()`.
- `env` specifies the environment variables that the child process will have access to. By default, child processes simply inherit `process.env`, but you can specify a different object if you want.
- `input` specifies a string or buffer of input data that should be used as the standard input to the child process. This option is only available to the synchronous functions that do not return a ChildProcess object.
- `maxBuffer` specifies the maximum number of bytes of output that will be collected by the `exec` functions. (It does not apply to `spawn()` and `fork()`, which use streams.) If a child process produces more output than this, it will be killed and will exit with an error.
- `shell` specifies the path to a shell executable or `true`. For child process functions that normally execute a shell command, this option allows you to specify which shell to use. For functions that do not normally use a shell, this option allows you to specify that a shell should be used (by setting the property to `true`) or to specify exactly which shell to use.
- `timeout` specifies the maximum number of milliseconds that the child process should be allowed to run. If it has not exited before this time elapses, it will be killed and will exit with an error. (This option applies to the `exec` functions but not to `spawn()` or `fork()`.)

- `uid` specifies the user ID (a number) under which the program should be run. If the parent process is running in a privileged account, it can use this option to run the child with reduced privileges.

16.10.2 exec() and execFile()

The `execSync()` and `execFileSync()` functions are, as their names indicate, synchronous: they block and do not return until the child process exits. Using these functions is a lot like typing Unix commands in a terminal window: they allow you to run a sequence of commands one at a time. But if you're writing a program that needs to accomplish a number of tasks, and those tasks don't depend on each other in any way, then you may want to parallelize them and run multiple commands at the same time. You can do this with the asynchronous functions `child_process.exec()` and `child_process.execFile()`.

`exec()` and `execFile()` are like their synchronous variants except that they return immediately with a ChildProcess object that represents the running child process, and they take an error-first callback as their final argument. The callback is invoked when the child process exits, and it is actually called with three arguments. The first is the error, if any; it will be `null` if the process terminated normally. The second argument is the collected output that was sent to the child's standard output stream. And the third argument is any output that was sent to the child's standard error stream.

The ChildProcess object returned by `exec()` and `execFile()` allows you to terminate the child process, and to write data to it (which it can then read from its standard input). We'll cover ChildProcess in more detail when we discuss the `child_process.spawn()` function.

If you plan to execute multiple child processes at the same time, then it may be easiest to use the "promisified" version of `exec()` which returns a Promise object which, if the child process exits without error, resolves to an object with `stdout` and `stderr` properties. Here, for example, is a function that takes an array of shell commands as its input and returns a Promise that resolves to the result of all of those commands:

```
const child_process = require("child_process");
const util = require("util");
const execP = util.promisify(child_process.exec);

function parallelExec(commands) {
    // Use the array of commands to create an array of Promises
    let promises = commands.map(command => execP(command, {encoding: "utf8"}));
    // Return a Promise that will fulfill to an array of the fulfillment
    // values of each of the individual promises. (Instead of returning objects
    // with stdout and stderr properties we just return the stdout value.)
    return Promise.all(promises)
```

```
        .then(outputs => outputs.map(out => out.stdout));
}

module.exports = parallelExec;
```

16.10.3 spawn()

The various `exec` functions described so far—both synchronous and asynchronous—are designed to be used with child processes that run quickly and do not produce a lot of output. Even the asynchronous `exec()` and `execFile()` are nonstreaming: they return the process output in a single batch, only after the process has exited.

The `child_process.spawn()` function allows you streaming access to the output of the child process, while the process is still running. It also allows you to write data to the child process (which will see that data as input on its standard input stream): this means it is possible to dynamically interact with a child process, sending it input based on the output it generates.

`spawn()` does not use a shell by default, so you must invoke it like `execFile()` with the executable to be run and a separate array of command-line arguments to pass to it. `spawn()` returns a ChildProcess object like `execFile()` does, but it does not take a callback argument. Instead of using a callback function, you listen to events on the ChildProcess object and on its streams.

The ChildProcess object returned by `spawn()` is an event emitter. You can listen for the "exit" event to be notified when the child process exits. A ChildProcess object also has three stream properties. `stdout` and `stderr` are Readable streams: when the child process writes to its stdout and its stderr streams, that output becomes readable through the ChildProcess streams. Note the inversion of the names here. In the child process, "stdout" is a Writable output stream, but in the parent process, the `stdout` property of a ChildProcess object is a Readable input stream.

Similarly, the `stdin` property of the ChildProcess object is a Writeable stream: anything you write to this stream becomes available to the child process on its standard input.

The ChildProcess object also defines a `pid` property that specifies the process id of the child. And it defines a `kill()` method that you can use to terminate a child process.

16.10.4 fork()

`child_process.fork()` is a specialized function for running a module of JavaScript code in a child Node process. `fork()` expects the same arguments as `spawn()`, but the first argument should specify the path to a file of JavaScript code instead of an executable binary file.

A child process created with `fork()` can communicate with the parent process via its standard input and standard output streams, as described in the previous section for `spawn()`. But in addition, `fork()` enables another, much easier, communication channel between the parent and child processes.

When you create a child process with `fork()`, you can use the `send()` method of the returned ChildProcess object to send a copy of an object to the child process. And you can listen for the "message" event on the ChildProcess to receive messages from the child. The code running in the child process can use `process.send()` to send a message to the parent and can listen for "message" events on `process` to receive messages from the parent.

Here, for example, is some code that uses `fork()` to create a child process, then sends that child a message and waits for a response:

```
const child_process = require("child_process");

// Start a new node process running the code in child.js in our directory
let child = child_process.fork(`${__dirname}/child.js`);

// Send a message to the child
child.send({x: 4, y: 3});

// Print the child's response when it arrives.
child.on("message", message => {
    console.log(message.hypotenuse); // This should print "5"
    // Since we only send one message we only expect one response.
    // After we receive it we call disconnect() to terminate the connection
    // between parent and child. This allows both processes to exit cleanly.
    child.disconnect();
});
```

And here is the code that runs in the child process:

```
// Wait for messages from our parent process
process.on("message", message => {
    // When we receive one, do a calculation and send the result
    // back to the parent.
    process.send({hypotenuse: Math.hypot(message.x, message.y)});
});
```

Starting child processes is an expensive operation, and the child process would have to be doing orders of magnitude more computation before it would make sense to use `fork()` and interprocess communication in this way. If you are writing a program that needs to be very responsive to incoming events and also needs to perform time-consuming computations, then you might consider using a separate child process to perform the computations so that they don't block the event loop and reduce the responsiveness of the parent process. (Though a thread—see §16.11—may be a better choice than a child process in this scenario.)

The first argument to `send()` will be serialized with `JSON.stringify()` and deserialized in the child process with `JSON.parse()`, so you should only include values that are supported by the JSON format. `send()` has a special second argument, however, that allows you to transfer Socket and Server objects (from the "net" module) to a child process. Network servers tend to be IO-bound rather than compute-bound, but if you have written a server that needs to do more computation than a single CPU can handle, and if you're running that server on a machine with multiple CPUs, then you could use `fork()` to create multiple child processes for handling requests. In the parent process, you might listen for "connection" events on your Server object, then get the Socket object from that "connection" event and `send()` it—using the special second argument—to one of the child processes to be handled. (Note that this is an unlikely solution to an uncommon scenario. Rather than writing a server that forks child processes, it is probably simpler to keep your server single-threaded and deploy multiple instances of it in production to handle the load.)

16.11 Worker Threads

As explained at the beginning of this chapter, Node's concurrency model is single-threaded and event-based. But in version 10 and later, Node does allow true multithreaded programming, with an API that closely mirrors the Web Workers API defined by web browsers (§15.13). Multithreaded programming has a well-deserved reputation for being difficult. This is almost entirely because of the need to carefully synchronize access by threads to shared memory. But JavaScript threads (in both Node and browsers) do not share memory by default, so the dangers and difficulties of using threads do not apply to these "workers" in JavaScript.

Instead of using shared memory, JavaScript's worker threads communicate by message passing. The main thread can send a message to a worker thread by calling the `postMessage()` method of the Worker object that represents that thread. The worker thread can receive messages from its parent by listening for "message" events. And workers can send messages to the main thread with their own version of `postMessage()`, which the parent can receive with its own "message" event handler. The example code will make it clear how this works.

There are three reasons why you might want to use worker threads in a Node application:

- If your application actually needs to do more computation than one CPU core can handle, then threads allow you to distribute work across the multiple cores, which have become commonplace on computers today. If you're doing scientific computing or machine learning or graphics processing in Node, then you may want to use threads simply to throw more computing power at your problem.

- Even if your application is not using the full power of one CPU, you may still want to use threads to maintain the responsiveness of the main thread. Consider a server that handles large but relatively infrequent requests. Suppose it gets only one request a second, but needs to spend about half a second of (blocking CPU-bound) computation to process each request. On average, it will be idle 50% of the time. But when two requests arrive within a few milliseconds of each other, the server will not even be able to begin a response to the second request until the computation of the first response is complete. Instead, if the server uses a worker thread to perform the computation, the server can begin the response to both requests immediately and provide a better experience for the server's clients. Assuming the server has more than one CPU core, it can also compute the body of both responses in parallel, but even if there is only a single core, using workers still improves the responsiveness.
- In general, workers allow us to turn blocking synchronous operations into non-blocking asynchronous operations. If you are writing a program that depends on legacy code that is unavoidably synchronous, you may be able to use workers to avoid blocking when you need to call that legacy code.

Worker threads are not nearly as heavyweight as child processes, but they are not lightweight. It does not generally make sense to create a worker unless you have significant work for it to do. And, generally speaking, if your program is not CPU-bound and is not having responsiveness problems, then you probably do not need worker threads.

16.11.1 Creating Workers and Passing Messages

The Node module that defines workers is known as "worker_threads." In this section we'll refer to it with the identifier `threads`:

```
const threads = require("worker_threads");
```

This module defines a Worker class to represent a worker thread, and you can create a new thread with the `threads.Worker()` constructor. The following code demonstrates using this constructor to create a worker, and shows how to pass messages from main thread to worker and from worker to main thread. It also demonstrates a trick that allows you to put the main thread code and the worker thread code in the same file.[2]

```
const threads = require("worker_threads");
```

2 It is often cleaner and simpler to define the worker code in a separate file. But this trick of having two threads run different sections of the same file blew my mind when I first encountered it for the Unix `fork()` system call. And I think it is worth demonstrating this technique simply for its strange elegance.

```
// The worker_threads module exports the boolean isMainThread property.
// This property is true when Node is running the main thread and it is
// false when Node is running a worker. We can use this fact to implement
// the main and worker threads in the same file.
if (threads.isMainThread) {
    // If we're running in the main thread, then all we do is export
    // a function. Instead of performing a computationally intensive
    // task on the main thread, this function passes the task to a worker
    // and returns a Promise that will resolve when the worker is done.
    module.exports = function reticulateSplines(splines) {
        return new Promise((resolve,reject) => {
            // Create a worker that loads and runs this same file of code.
            // Note the use of the special __filename variable.
            let reticulator = new threads.Worker(__filename);

            // Pass a copy of the splines array to the worker
            reticulator.postMessage(splines);

            // And then resolve or reject the Promise when we get
            // a message or error from the worker.
            reticulator.on("message", resolve);
            reticulator.on("error", reject);
        });
    };
} else {
    // If we get here, it means we're in the worker, so we register a
    // handler to get messages from the main thread. This worker is designed
    // to only receive a single message, so we register the event handler
    // with once() instead of on(). This allows the worker to exit naturally
    // when its work is complete.
    threads.parentPort.once("message", splines => {
        // When we get the splines from the parent thread, loop
        // through them and reticulate all of them.
        for(let spline of splines) {
            // For the sake of example, assume that spline objects usually
            // have a reticulate() method that does a lot of computation.
            spline.reticulate ? spline.reticulate() : spline.reticulated = true;
        }

        // When all the splines have (finally!) been reticulated
        // pass a copy back to the main thread.
        threads.parentPort.postMessage(splines);
    });
}
```

The first argument to the `Worker()` constructor is the path to a file of JavaScript code that is to run in the thread. In the preceding code, we used the predefined `__filename` identifier to create a worker that loads and runs the same file as the main thread. In general, though, you will be passing a file path. Note that if you specify a relative path, it is relative to `process.cwd()`, not relative to the currently running module. If

you want a path relative to the current module, use something like `path.resolve(__dirname, 'workers/reticulator.js')`.

The `Worker()` constructor can also accept an object as its second argument, and the properties of this object provide optional configuration for the worker. We'll cover a number of these options later, but for now note that if you pass `{eval: true}` as the second argument, then the first argument to `Worker()` is interpreted as a string of JavaScript code to be evaluated instead of a filename:

```
new threads.Worker(`
    const threads = require("worker_threads");
    threads.parentPort.postMessage(threads.isMainThread);
`, {eval: true}).on("message", console.log);  // This will print "false"
```

Node makes a copy of the object passed to `postMessage()` rather than sharing it directly with the worker thread. This prevents the worker thread and the main thread from sharing memory. You might expect that this copying would be done with `JSON.stringify()` and `JSON.parse()` (§11.6). But in fact, Node borrows a more robust technique known as the structured clone algorithm from web browsers.

The structured clone algorithm enables serialization of most JavaScript types, including Map, Set, Date, and RegExp objects and typed arrays, but it cannot, in general, copy types defined by the Node host environment, such as sockets and streams. Note, however, that Buffer objects are partially supported: if you pass a Buffer to `postMessage()` it will be received as a Uint8Array, and can be converted back into a Buffer with `Buffer.from()`. Read more about the structured clone algorithm in "The Structured Clone Algorithm" on page 513.

16.11.2 The Worker Execution Environment

For the most part, JavaScript code in a Node worker thread runs just like it would in Node's main thread. There are a few differences that you should be aware of, and some of these differences involve properties of the optional second argument to the `Worker()` constructor:

- As we've seen, `threads.isMainThread` is `true` in the main thread but is always `false` in any worker thread.
- In a worker thread, you can use `threads.parentPort.postMessage()` to send a message to the parent thread and `threads.parentPort.on` to register event handlers for messages from the parent thread. In the main thread, `threads.parentPort` is always `null`.
- In a worker thread, `threads.workerData` is set to a copy of the `workerData` property of the second argument to the `Worker()` constructor. In the main thread, this property is always `null`. You can use this `workerData` property to pass an initial

message to the worker that will be available as soon as it starts so that the worker does not have to wait for a "message" event before it can start doing work.

- By default, `process.env` in a worker thread is a copy of `process.env` in the parent thread. But the parent thread can specify a custom set of environment variables by setting the `env` property of the second argument to the `Worker()` constructor. As a special (and potentially dangerous) case, the parent thread can set the `env` property to `threads.SHARE_ENV`, which will cause the two threads to share a single set of environment variables so that a change in one thread is visible in the other.
- By default, the `process.stdin` stream in a worker never has any readable data on it. You can change this default by passing `stdin: true` in the second argument to the `Worker()` constructor. If you do that, then the `stdin` property of the Worker object is a Writable stream. Any data that the parent writes to `worker.stdin` becomes readable on `process.stdin` in the worker.
- By default, the `process.stdout` and `process.stderr` streams in the worker are simply piped to the corresponding streams in the parent thread. This means, for example, that `console.log()` and `console.error()` produce output in exactly the same way in a worker thread as they do in the main thread. You can override this default by passing `stdout:true` or `stderr:true` in the second argument to the `Worker()` constructor. If you do this, then any output the worker writes to those streams becomes readable by the parent thread on the `worker.stdout` and `worker.stderr` threads. (There is a potentially confusing inversion of stream directions here, and we saw the same thing with with child processes earlier in the chapter: the output streams of a worker thread are input streams for the parent thread, and the input stream of a worker is an output stream for the parent.)
- If a worker thread calls `process.exit()`, only the thread exits, not the entire process.
- Worker threads are not allowed to change shared state of the process they are part of. Functions like `process.chdir()` and `process.setuid()` will throw exceptions when invoked from a worker.
- Operating system signals (like `SIGINT` and `SIGTERM`) are only delivered to the main thread; they cannot be received or handled in worker threads.

16.11.3 Communication Channels and MessagePorts

When a new worker thread is created, a communication channel is created along with it that allows messages to be passed back and forth between the worker and the parent thread. As we've seen, the worker thread uses `threads.parentPort` to send and

receive messages to and from the parent thread, and the parent thread uses the Worker object to send and receive messages to and from the worker thread.

The worker thread API also allows the creation of custom communication channels using the MessageChannel API defined by web browsers and covered in §15.13.5. If you have read that section, much of what follows will sound familiar to you.

Suppose a worker needs to handle two different kinds of messages sent by two different modules in the main thread. These two different modules could both share the default channel and send messages with `worker.postMessage()`, but it would be cleaner if each module has its own private channel for sending messages to the worker. Or consider the case where the main thread creates two independent workers. A custom communication channel can allow the two workers to communicate directly with each other instead of having to send all their messages via the parent.

Create a new message channel with the `MessageChannel()` constructor. A MessageChannel object has two properties, named `port1` and `port2`. These properties refer to a pair of MessagePort objects. Calling `postMessage()` on one of the ports will cause a "message" event to be generated on the other with a structured clone of the Message object:

```
const threads = require("worker_threads");
let channel = new threads.MessageChannel();
channel.port2.on("message", console.log);  // Log any messages we receive
channel.port1.postMessage("hello");        // Will cause "hello" to be printed
```

You can also call `close()` on either port to break the connection between the two ports and to signal that no more messages will be exchanged. When `close()` is called on either port, a "close" event is delivered to both ports.

Note that the code example above creates a pair of MessagePort objects and then uses those objects to transmit a message within the main thread. In order to use custom communication channels with workers, we must transfer one of the two ports from the thread in which it is created to the thread in which it will be used. The next section explains how to do this.

16.11.4 Transferring MessagePorts and Typed Arrays

The `postMessage()` function uses the structured clone algorithm, and as we've noted, it cannot copy objects like SSockets and Streams. It can handle MessagePort objects, but only as a special case using a special technique. The `postMessage()` method (of a Worker object, of `threads.parentPort`, or of any MessagePort object) takes an optional second argument. This argument (called `transferList`) is an array of objects that are to be transferred between threads rather than being copied.

A MessagePort object cannot be copied by the structured clone algorithm, but it can be transferred. If the first argument to `postMessage()` has included one or more

MessagePorts (nested arbitrarily deeply within the Message object), then those MessagePort objects must also appear as members of the array passed as the second argument. Doing this tells Node that it does not need to make a copy of the MessagePort, and can instead just give the existing object to the other thread. The key thing to understand, however, about transferring values between threads is that once a value is transferred, it can no longer be used in the thread that called `postMessage()`.

Here is how you might create a new MessageChannel and transfer one of its MessagePorts to a worker:

```
// Create a custom communication channel
const threads = require("worker_threads");
let channel = new threads.MessageChannel();

// Use the worker's default channel to transfer one end of the new
// channel to the worker. Assume that when the worker receives this
// message it immediately begins to listen for messages on the new channel.
worker.postMessage({ command: "changeChannel", data: channel.port1 },
                   [ channel.port1 ]);

// Now send a message to the worker using our end of the custom channel
channel.port2.postMessage("Can you hear me now?");

// And listen for responses from the worker as well
channel.port2.on("message", handleMessagesFromWorker);
```

MessagePort objects are not the only ones that can be transferred. If you call `postMessage()` with a typed array as the message (or with a message that contains one or more typed arrays nested arbitrarily deep within the message), that typed array (or those typed arrays) will simply be copied by the structured clone algorithm. But typed arrays can be large; for example, if you are using a worker thread to do image processing on millions of pixels. So for efficiency, `postMessage()` also gives us the option to transfer typed arrays rather than copying them. (Threads share memory by default. Worker threads in JavaScript generally avoid shared memory, but when we allow this kind of controlled transfer, it can be done very efficiently.) What makes this safe is that when a typed array is transferred to another thread, it becomes unusable in the thread that transferred it. In the image-processing scenario, the main thread could transfer the pixels of an image to the worker thread, and then the worker thread could transfer the processed pixels back to the main thread when it was done. The memory would not need to be copied, but it would never be accessible by two threads at once.

To transfer a typed array instead of copying it, include the ArrayBuffer that backs the array in the second argument to `postMessage()`:

```
let pixels = new Uint32Array(1024*1024);  // 4 megabytes of memory

// Assume we read some data into this typed array, and then transfer the
```

```
// pixels to a worker without copying. Note that we don't put the array
// itself in the transfer list, but the array's Buffer object instead.
worker.postMessage(pixels, [ pixels.buffer ]);
```

As with transferred MessagePorts, a transferred typed array becomes unusable once transferred. No exceptions are thrown if you attempt to use a MessagePort or typed array that has been transferred; these objects simply stop doing anything when you interact with them.

16.11.5 Sharing Typed Arrays Between Threads

In addition to transferring typed arrays between threads, it is actually possible to share a typed array between threads. Simply create a SharedArrayBuffer of the desired size and then use that buffer to create a typed array. When a typed array that is backed by a SharedArrayBuffer is passed via `postMessage()`, the underlying memory will be shared between the threads. You should not include the shared buffer in the second argument to `postMessage()` in this case.

You really should not do this, however, because JavaScript was never designed with thread safety in mind and multithreaded programming is very difficult to get right. (And this is why SharedArrayBuffer was not covered in §11.2: it is a niche feature that is difficult to get right.) Even the simple `++` operator is not thread-safe because it needs to read a value, increment it, and write it back. If two threads are incrementing a value at the same time, it will often only be incremented once, as the following code demonstrates:

```
const threads = require("worker_threads");

if (threads.isMainThread) {
    // In the main thread, we create a shared typed array with
    // one element. Both threads will be able to read and write
    // sharedArray[0] at the same time.
    let sharedBuffer = new SharedArrayBuffer(4);
    let sharedArray = new Int32Array(sharedBuffer);

    // Now create a worker thread, passing the shared array to it with
    // as its initial workerData value so we don't have to bother with
    // sending and receiving a message
    let worker = new threads.Worker(__filename, { workerData: sharedArray });

    // Wait for the worker to start running and then increment the
    // shared integer 10 million times.
    worker.on("online", () => {
        for(let i = 0; i < 10_000_000; i++) sharedArray[0]++;

        // Once we're done with our increments, we start listening for
        // message events so we know when the worker is done.
        worker.on("message", () => {
            // Although the shared integer has been incremented
```

```
            // 20 million times, its value will generally be much less.
            // On my computer the final value is typically under 12 million.
            console.log(sharedArray[0]);
        });
    });
} else {
    // In the worker thread, we get the shared array from workerData
    // and then increment it 10 million times.
    let sharedArray = threads.workerData;
    for(let i = 0; i < 10_000_000; i++) sharedArray[0]++;
    // When we're done incrementing, let the main thread know
    threads.parentPort.postMessage("done");
}
```

One scenario in which it might be reasonable to use a SharedArrayBuffer is when the two threads operate on entirely separate sections of the shared memory. You might enforce this by creating two typed arrays that serve as views of nonoverlapping regions of the shared buffer, and then have your two threads use those two separate typed arrays. A parallel merge sort could be done like this: one thread sorts the bottom half of an array and the other thread sorts the top half, for example. Or some kinds of image-processing algorithms are also suitable for this approach: multiple threads working on disjoint regions of the image.

If you really must allow multiple threads to access the same region of a shared array, you can take one step toward thread safety with the functions defined by the Atomics object. Atomics was added to JavaScript when SharedArrayBuffer was to define atomic operations on the elements of a shared array. For example, the `Atomics.add()` function reads the specified element of a shared array, adds a specified value to it, and writes the sum back into the array. It does this atomically as if it was a single operation, and ensures that no other thread can read or write the value while the operation is taking place. `Atomics.add()` allows us to rewrite the parallel increment code we just looked at and get the correct result of 20 million increments of a shared array element:

```
const threads = require("worker_threads");

if (threads.isMainThread) {
    let sharedBuffer = new SharedArrayBuffer(4);
    let sharedArray = new Int32Array(sharedBuffer);
    let worker = new threads.Worker(__filename, { workerData: sharedArray });

    worker.on("online", () => {
        for(let i = 0; i < 10_000_000; i++) {
            Atomics.add(sharedArray, 0, 1);  // Threadsafe atomic increment
        }

        worker.on("message", (message) => {
            // When both threads are done, use a threadsafe function
            // to read the shared array and confirm that it has the
```

```
            // expected value of 20,000,000.
            console.log(Atomics.load(sharedArray, 0));
        });
    });
} else {
    let sharedArray = threads.workerData;
    for(let i = 0; i < 10_000_000; i++) {
        Atomics.add(sharedArray, 0, 1);      // Threadsafe atomic increment
    }
    threads.parentPort.postMessage("done");
}
```

This new version of the code correctly prints the number 20,000,000. But it is about nine times slower than the incorrect code it replaces. It would be much simpler and much faster to just do all 20 million increments in one thread. Also note that atomic operations may be able to ensure thread safety for image-processing algorithms for which each array element is a value entirely independent of all other values. But in most real-world programs, multiple array elements are often related to one another and some kind of higher-level thread synchronization is required. The low-level `Atomics.wait()` and `Atomics.notify()` function can help with this, but a discussion of their use is out of scope for this book.

16.12 Summary

Although JavaScript was created to run in web browsers, Node has made JavaScript into a general-purpose programming language. It is particularly popular for implementing web servers, but its deep bindings to the operating system mean that it is also a good alternative to shell scripts.

The most important topics covered in this long chapter include:

- Node's asynchronous-by-default APIs and its single-threaded, callback, and event-based style of concurrency.
- Node's fundamental datatypes, buffers, and streams.
- Node's "fs" and "path" modules for working with the filesystem.
- Node's "http" and "https" modules for writing HTTP clients and servers.
- Node's "net" module for writing non-HTTP clients and servers.
- Node's "child_process" module for creating and communicating with child processes.
- Node's "worker_threads" module for true multithreaded programming using message-passing instead of shared memory.

CHAPTER 17

JavaScript Tools and Extensions

Congratulations on reaching the final chapter of this book. If you have read everything that comes before, you now have a detailed understanding of the JavaScript language and know how to use it in Node and in web browsers. This chapter is a kind of graduation present: it introduces a handful of important programming tools that many JavaScript programmers find useful, and also describes two widely used extensions to the core JavaScript language. Whether or not you choose to use these tools and extensions for your own projects, you are almost certain to see them used in other projects, so it is important to at least know what they are.

The tools and language extensions covered in this chapter are:

- ESLint for finding potential bugs and style problems in your code.
- Prettier for formatting your JavaScript code in a standardized way.
- Jest as an all-in-one solution for writing JavaScript unit tests.
- npm for managing and installing the software libraries that your program depends on.
- Code-bundling tools—like webpack, Rollup, and Parcel—that convert your modules of JavaScript code into a single bundle for use on the web.
- Babel for translating JavaScript code that uses brand-new language features (or that uses language extensions) into JavaScript code that can run in current web browsers.
- The JSX language extension (used by the React framework) that allows you to describe user interfaces using JavaScript expressions that look like HTML markup.

- The Flow language extension (or the similar TypeScript extension) that allows you to annotate your JavaScript code with types and check your code for type safety.

This chapter does not document these tools and extensions in any comprehensive way. The goal is simply to explain them in enough depth that you can understand why they are useful and when you might want to use them. Everything covered in this chapter is widely used in the JavaScript programming world, and if you do decide to adopt a tool or extension, you'll find lots of documentation and tutorials online.

17.1 Linting with ESLint

In programming, the term *lint* refers to code that, while technically correct, is unsightly, or a possible bug, or suboptimal in some way. A *linter* is a tool for detecting lint in your code, and *linting* is the process of running a linter on your code (and then fixing your code to remove the lint so that the linter no longer complains).

The most commonly used linter for JavaScript today is ESLint. If you run it and then take the time to actually fix the issues it points out, it will make your code cleaner and less likely to have bugs. Consider the following code:

```
var x = 'unused';

export function factorial(x) {
    if (x == 1) {
      return 1;
    } else {
        return x * factorial(x-1)
    }
}
```

If you run ESLint on this code, you might get output like this:

```
$ eslint code/ch17/linty.js

code/ch17/linty.js
  1:1   error    Unexpected var, use let or const instead      no-var
  1:5   error    'x' is assigned a value but never used        no-unused-vars
  1:9   warning  Strings must use doublequote                  quotes
  4:11  error    Expected '===' and instead saw '=='           eqeqeq
  5:1   error    Expected indentation of 8 spaces but found 6  indent
  7:28  error    Missing semicolon                             semi

✖ 6 problems (5 errors, 1 warning)
  3 errors and 1 warning potentially fixable with the `--fix` option.
```

Linters can seem nitpicky sometimes. Does it really matter whether we used double quotes or single quotes for our strings? On the other hand, getting indentation right is important for readability, and using `===` and `let` instead of `==` and `var` protects you

from subtle bugs. And unused variables are dead weight in your code—there is no reason to keep those around.

ESLint defines many linting rules and has an ecosystem of plug-ins that add many more. But ESLint is fully configurable, and you can define a configuration file that tunes ESLint to enforce exactly the rules you want and only those rules.

17.2 JavaScript Formatting with Prettier

One of the reasons that some projects use linters is to enforce a consistent coding style so that when a team of programmers is working on a shared codebase, they use compatible code conventions. This includes code indentation rules, but can also include things like what kind of quotation marks are preferred and whether there should be a space between the `for` keyword and the open parenthesis that follows it.

A modern alternative to enforcing code formatting rules via a linter is to adopt a tool like Prettier to automatically parse and reformat all of your code.

Suppose you have written the following function, which works, but is formatted unconventionally:

```
function factorial(x)
{
         if(x===1){return 1}
           else{return x*factorial(x-1)}
}
```

Running Prettier on this code fixes the indentation, adds missing semicolons, adds spaces around binary operators and inserts line breaks after `{` and before `}`, resulting in much more conventional-looking code:

```
$ prettier factorial.js
function factorial(x) {
  if (x === 1) {
    return 1;
  } else {
    return x * factorial(x - 1);
  }
}
```

If you invoke Prettier with the `--write` option, it will simply reformat the specified file in place rather than printing a reformatted version. If you use `git` to manage your source code, you can invoke Prettier with the `--write` option in a commit hook so that code is automatically formatted before being checked in.

Prettier is particularly powerful if you configure your code editor to run it automatically every time you save a file. I find it liberating to write sloppy code and see it fixed automatically for me.

Prettier is configurable, but it only has a few options. You can select the maximum line length, the indentation amount, whether semicolons should be used, whether strings should be single- or double-quoted, and a few other things. In general, Prettier's default options are quite reasonable. The idea is that you just adopt Prettier for your project and then never have to think about code formatting again.

Personally, I really like using Prettier on JavaScript projects. I have not used it for the code in this book, however, because in much of my code I rely on careful hand formatting to align my comments vertically, and Prettier messes them up.

17.3 Unit Testing with Jest

Writing tests is an important part of any nontrivial programming project. Dynamic languages like JavaScript support testing frameworks that dramatically reduce the effort required to write tests, and almost make test writing fun! There are a lot of test tools and libraries for JavaScript, and many are written in a modular way so that it is possible to pick one library as your test runner, another library for assertions, and a third for mocking. In this section, however, we'll describe Jest, which is a popular framework that includes everything you need in a single package.

Suppose you've written the following function:

```
const getJSON = require("./getJSON.js");

/**
 * getTemperature() takes the name of a city as its input, and returns
 * a Promise that will resolve to the current temperature of that city,
 * in degrees Fahrenheit. It relies on a (fake) web service that returns
 * world temperatures in degrees Celsius.
 */
module.exports = async function getTemperature(city) {
    // Get the temperature in Celsius from the web service
    let c = await getJSON(
        `https://globaltemps.example.com/api/city/${city.toLowerCase()}`
    );
    // Convert to Fahrenheit and return that value.
    return (c * 5 / 9) + 32;  // TODO: double-check this formula
};
```

A good set of tests for this function might verify that `getTemperature()` is fetching the right URL, and that it is converting temperature scales correctly. We can do this with a Jest-based test like the following. This code defines a mock implementation of `getJSON()` so that the test does not actually make a network request. And because `getTemperature()` is an async function, the tests are async as well—it can be tricky to test asynchronous functions, but Jest makes it relatively easy:

```
// Import the function we are going to test
const getTemperature = require("./getTemperature.js");
```

```
// And mock the getJSON() module that getTemperature() depends on
jest.mock("./getJSON");
const getJSON = require("./getJSON.js");

// Tell the mock getJSON() function to return an already resolved Promise
// with fulfillment value 0.
getJSON.mockResolvedValue(0);

// Our set of tests for getTemperature() begins here
describe("getTemperature()", () => {
    // This is the first test. We're ensuring that getTemperature() calls
    // getJSON() with the URL that we expect
    test("Invokes the correct API", async () => {
        let expectedURL = "https://globaltemps.example.com/api/city/vancouver";
        let t = await(getTemperature("Vancouver"));
        // Jest mocks remember how they were called, and we can check that.
        expect(getJSON).toHaveBeenCalledWith(expectedURL);
    });

    // This second test verifies that getTemperature() converts
    // Celsius to Fahrenheit correctly
    test("Converts C to F correctly", async () => {
        getJSON.mockResolvedValue(0);               // If getJSON returns 0C
        expect(await getTemperature("x")).toBe(32);  // We expect 32F

        // 100C should convert to 212F
        getJSON.mockResolvedValue(100);             // If getJSON returns 100C
        expect(await getTemperature("x")).toBe(212); // We expect 212F
    });
});
```

With the test written, we can use the `jest` command to run it, and we discover that one of our tests fails:

```
$ jest getTemperature
 FAIL  ch17/getTemperature.test.js
  getTemperature()
    ✓ Invokes the correct API (4ms)
    ✕ Converts C to F correctly (3ms)

  ● getTemperature() › Converts C to F correctly

    expect(received).toBe(expected) // Object.is equality

    Expected: 212
    Received: 87.55555555555556

      29 |         // 100C should convert to 212F
      30 |         getJSON.mockResolvedValue(100); // If getJSON returns 100C
    > 31 |         expect(await getTemperature("x")).toBe(212); // Expect 212F
         |                                           ^
      32 |     });
```

```
      33 | });
      34 |

      at Object.<anonymous> (ch17/getTemperature.test.js:31:43)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 passed, 2 total
Snapshots:   0 total
Time:        1.403s
Ran all test suites matching /getTemperature/i.
```

Our `getTemperature()` implementation is using the wrong formula for converting C to F. It multiplies by 5 and divides by 9 rather than multiplying by 9 and dividing by 5. If we fix the code and run Jest again, we can see the tests pass. And, as a bonus, if we add the `--coverage` argument when we invoke `jest`, it will compute and display the code coverage for our tests:

```
$ jest --coverage getTemperature
 PASS  ch17/getTemperature.test.js
  getTemperature()
    ✓ Invokes the correct API (3ms)
    ✓ Converts C to F correctly (1ms)

------------------|--------|---------|---------|---------|------------------|
File              | % Stmts| % Branch|  % Funcs|  % Lines| Uncovered Line #s|
------------------|--------|---------|---------|---------|------------------|
All files         |   71.43|      100|    33.33|    83.33|                  |
 getJSON.js       |   33.33|      100|        0|       50|                2|
 getTemperature.js|     100|      100|      100|      100|                  |
------------------|--------|---------|---------|---------|------------------|
Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        1.508s
Ran all test suites matching /getTemperature/i.
```

Running our test gave us 100% code coverage for the module we were testing, which is exactly what we wanted. It only gave us partial coverage of `getJSON()`, but we mocked that module and were not trying to test it, so that is expected.

17.4 Package Management with npm

In modern software development, it is common for any nontrivial program that you write to depend on third-party software libraries. If you're writing a web server in Node, for example, you might be using the Express framework. And if you're creating a user interface to be displayed in a web browser, you might use a frontend framework like React or LitElement or Angular. A package manager makes it easy to find and install third-party packages like these. Just as importantly, a package manager keeps track of what packages your code depends on and saves this information into a

file so that when someone else wants to try your program, they can download your code and your list of dependencies, then use their own package manager to install all the third-party packages that your code needs.

npm is the package manager that is bundled with Node, and was introduced in §16.1.5. It is just as useful for client-side JavaScript programming as it is for server-side programming with Node, however.

If you are trying out someone else's JavaScript project, then one of the first things you will often do after downloading their code is to type `npm install`. This reads the dependencies listed in the *package.json* file and downloads the third-party packages that the project needs and saves them in a *node_modules/* directory.

You can also type `npm install <package-name>` to install a particular package to your project's *node_modules/* directory:

```
$ npm install express
```

In addition to installing the named package, npm also makes a record of the dependency in the *package.json* file for the project. Recording dependencies in this way is what allows others to install those dependencies simply by typing `npm install`.

The other kind of dependency is on developer tools that are needed by developers who want to work on your project, but aren't actually needed to run the code. If a project uses Prettier, for example, to ensure that all of its code is consistently formatted, then Prettier is a "dev dependency," and you can install and record one of these with `--save-dev`:

```
$ npm install --save-dev prettier
```

Sometimes you might want to install developer tools globally so that they are accessible anywhere even for code that is not part of a formal project with a *package.json* file and a *node_modules/* directory. For that you can use the `-g` (for global) option:

```
$ npm install -g eslint jest
/usr/local/bin/eslint -> /usr/local/lib/node_modules/eslint/bin/eslint.js
/usr/local/bin/jest -> /usr/local/lib/node_modules/jest/bin/jest.js
+ jest@24.9.0
+ eslint@6.7.2
added 653 packages from 414 contributors in 25.596s

$ which eslint
/usr/local/bin/eslint
$ which jest
/usr/local/bin/jest
```

In addition to the "install" command, npm supports "uninstall" and "update" commands, which do what their names say. npm also has an interesting "audit" command that you can use to find and fix security vulnerabilities in your dependencies:

```
$ npm audit --fix

                        === npm audit security report ===

found 0 vulnerabilities
 in 876354 scanned packages
```

When you install a tool like ESLint locally for a project, the eslint script winds up in *./node_modules/.bin/eslint*, which makes the command awkward to run. Fortunately, npm is bundled with a command known as "npx," which you can use to run locally installed tools with commands like `npx eslint` or `npx jest`. (And if you use npx to invoke a tool that has not been installed yet, it will install it for you.)

The company behind npm also maintains the *https://npmjs.com* package repository, which holds hundreds of thousands of open source packages. But you don't have to use the npm package manager to access this repository of packages. Alternatives include yarn and pnpm.

17.5 Code Bundling

If you are writing a large JavaScript program to run in web browsers, you will probably want to use a code-bundling tool, especially if you use external libraries that are delivered as modules. Web developers have been using ES6 modules (§10.3) for years, since well before the `import` and `export` keywords were supported on the web. In order to do this, programmers use a code-bundler tool that starts at the main entry point (or entry points) of the program and follows the tree of `import` directives to find all modules that the program depends on. It then combines all of those individual module files into a single bundle of JavaScript code and rewrites the `import` and `export` directives to make the code work in this new form. The result is a single file of code that can be loaded into a web browser that does not support modules.

ES6 modules are nearly universally supported by web browsers today, but web developers still tend to use code bundlers, at least when releasing production code. Developers find that user experience is best when a single medium-sized bundle of code is loaded when a user first visits a website than when many small modules are loaded.

Web performance is a notoriously tricky topic and there are lots of variables to consider, including ongoing improvements by browser vendors, so the only way to be sure of the fastest way to load your code is by testing thoroughly and measuring carefully. Keep in mind that there is one variable that is completely under your control: code size. Less JavaScript code will always load and run faster than more JavaScript code!

There are a number of good JavaScript bundler tools available. Commonly used bundlers include webpack, Rollup and Parcel. The basic features of bundlers are more or less the same, and they are differentiated based on how configurable they are or how easy they are to use. Webpack has been around for a long time, has a large ecosystem of plug-ins, is highly configurable, and can support older nonmodule libraries. But it can also be complex and hard to configure. At the other end of the spectrum is Parcel which is intended as a zero-configuration alternative that simply does the right thing.

In addition to performing basic bundling, bundler tools can also provide some additional features:

- Some programs have more than one entry point. A web application with multiple pages, for example, could be written with a different entry point for each page. Bundlers generally allow you to create one bundle per entry point or to create a single bundle that supports multiple entry points.
- Programs can use `import()` in its functional form (§10.3.6) instead of its static form to dynamically load modules when they are actually needed rather than statically loading them at program startup time. Doing this is often a good way to improve the startup time for your program. Bundler tools that support `import()` may be able to produce multiple output bundles: one to load at startup time, and one or more that are loaded dynamically when needed. This can work well if there are only a few calls to `import()` in your program and they load modules with relatively disjoint sets of dependencies. If the dynamically loaded modules share dependencies then it becomes tricky to figure out how many bundles to produce, and you are likely to have to manually configure your bundler to sort this out.
- Bundlers can generally output a *source map* file that defines a mapping between the lines of code in the bundle and the corresponding lines in the original source files. This allows browser developer tools to automatically display JavaScript errors at their original unbundled locations.
- Sometimes when you import a module into your program, you only use a few of its features. A good bundler tool can analyze the code to determine which parts are unused and can be omitted from the bundles. This feature goes by the whimsical name of "tree-shaking."
- Bundlers typically have a plug-in–based architecture and support plug-ins that allow importing and bundling "modules" that are not actually files of JavaScript code. Suppose that your program includes a large JSON-compatible data structure. Code bundlers can be configured to allow you to move that data structure into a separate JSON file and then import it into your program with a declaration like `import widgets from "./big-widget-list.json"`. Similarly, web developers who embed CSS into their JavaScript programs can use bundler plug-ins that

allow them to import CSS files with an `import` directive. Note, however, that if you import anything other than a JavaScript file, you are using a nonstandard JavaScript extension and making your code dependent on the bundler tool.

- In a language like JavaScript that does not require compilation, running a bundler tool feels like a compilation step, and it is frustrating to have to run a bundler after every code edit before you can run the code in your browser. Bundlers typically support filesystem watchers that detect edits to any files in a project directory and automatically regenerate the necessary bundles. With this feature in place you can typically save your code and then immediately reload your web browser window to try it out.
- Some bundlers also support a "hot module replacement" mode for developers where each time a bundle is regenerated, it is automatically loaded into the browser. When this works, it is a magical experience for developers, but there are some tricks going on under the hood to make it work, and it is not suitable for all projects.

17.6 Transpilation with Babel

Babel is a tool that compiles JavaScript written using modern language features into JavaScript that does not use those modern language features. Because it compiles JavaScript to JavaScript, Babel is sometimes called a "transpiler." Babel was created so that web developers could use the new language features of ES6 and later while still targeting web browsers that only supported ES5.

Language features such as the `**` exponentiation operator and arrow functions can be transformed relatively easily into `Math.pow()` and `function` expressions. Other language features, such as the `class` keyword, require much more complex transformations, and, in general, the code output by Babel is not meant to be human readable. Like bundler tools, however, Babel can produce source maps that map transformed code locations back to their original source locations, and this helps dramatically when working with transformed code.

Browser vendors are doing a better job of keeping up with the evolution of the JavaScript language, and there is much less need today to compile away arrow functions and class declarations. Babel can still help when you want to use the very latest features like underscore separators in numeric literals.

Like most of the other tools described in this chapter, you can install Babel with npm and run it with npx. Babel reads a *.babelrc* configuration file that tells it how you would like your JavaScript code transformed. Babel defines "presets" that you can choose from depending on which language extensions you want to use and how aggressively you want to transform standard language features. One of Babel's

interesting presets is for code compression by minification (stripping comments and whitespace, renaming variables, and so on).

If you use Babel and a code-bundling tool, you may be able to set up the code bundler to automatically run Babel on your JavaScript files as it builds the bundle for you. If so, this can be a convenient option because it simplifies the process of producing runnable code. Webpack, for example, supports a "babel-loader" module that you can install and configure to run Babel on each JavaScript module as it is bundled up.

Even though there is less need to transform the core JavaScript language today, Babel is still commonly used to support nonstandard extensions to the language, and we'll describe two of these language extensions in the sections that follow.

17.7 JSX: Markup Expressions in JavaScript

JSX is an extension to core JavaScript that uses HTML-style syntax to define a tree of elements. JSX is most closely associated with the React framework for user interfaces on the web. In React, the trees of elements defined with JSX are ultimately rendered into a web browser as HTML. Even if you have no plans to use React yourself, its popularity means that you are likely to see code that uses JSX. This section explains what you need to know to make sense of of it. (This section is about the JSX language extension, not about React, and it explains only enough of React to provide context for the JSX syntax.)

You can think of a JSX element as a new type of JavaScript expression syntax. JavaScript string literals are delimited with quotation marks, and regular expression literals are delimited with slashes. In the same way, JSX expression literals are delimited with angle brackets. Here is a very simple one:

```
let line = <hr/>;
```

If you use JSX, you will need to use Babel (or a similar tool) to compile JSX expressions into regular JavaScript. The transformation is simple enough that some developers choose to use React without using JSX. Babel transforms the JSX expression in this assignment statement into a simple function call:

```
let line = React.createElement("hr", null);
```

JSX syntax is HTML-like, and like HTML elements, React elements can have attributes like these:

```
let image = <img src="logo.png" alt="The JSX logo" hidden/>;
```

When an element has one or more attributes, they become properties of an object passed as the second argument to `createElement()`:

```
let image = React.createElement("img", {
              src: "logo.png",
              alt: "The JSX logo",
```

```
    hidden: true
  });
```

Like HTML elements, JSX elements can have strings and other elements as children. Just as JavaScript's arithmetic operators can be used to write arithmetic expressions of arbitrary complexity, JSX elements can also be nested arbitrarily deeply to create trees of elements:

```
let sidebar = (
  <div className="sidebar">
    <h1>Title</h1>
    <hr/>
    <p>This is the sidebar content</p>
  </div>
);
```

Regular JavaScript function call expressions can also be nested arbitrarily deeply, and these nested JSX expressions translate into a set of nested `createElement()` calls. When an JSX element has children, those children (which are typically strings and other JSX elements) are passed as the third and subsequent arguments:

```
let sidebar = React.createElement(
    "div", { className: "sidebar"},  // This outer call creates a <div>
    React.createElement("h1", null,  // This is the first child of the <div/>
                        "Title"),    // and its own first child.
    React.createElement("hr", null), // The second child of the <div/>.
    React.createElement("p", null,   // And the third child.
                        "This is the sidebar content"));
```

The value returned by `React.createElement()` is an ordinary JavaScript object that is used by React to render output in a browser window. Since this section is about the JSX syntax and not about React, we're not going to go into any detail about the returned Element objects or the rendering process. It is worth noting that you can configure Babel to compile JSX elements to invocations of a different function, so if you think that JSX syntax would be a useful way to express other kinds of nested data structures, you can adopt it for your own non-React uses.

An important feature of JSX syntax is that you can embed regular JavaScript expressions within JSX expressions. Within a JSX expression, text within curly braces is interpreted as plain JavaScript. These nested expressions are allowed as attribute values and as child elements. For example:

```
function sidebar(className, title, content, drawLine=true) {
  return (
    <div className={className}>
      <h1>{title}</h1>
      { drawLine && <hr/> }
      <p>{content}</p>
    </div>
  );
}
```

The `sidebar()` function returns a JSX element. It takes four arguments that it uses within the JSX element. The curly brace syntax may remind you of template literals that use `${}` to include JavaScript expressions within strings. Since we know that JSX expressions compile into function invocations, it should not be surprising that arbitrary JavaScript expressions can be included because function invocations can be written with arbitrary expressions as well. This example code is translated by Babel into the following:

```
function sidebar(className, title, content, drawLine=true) {
  return React.createElement("div", { className: className },
                             React.createElement("h1", null, title),
                             drawLine && React.createElement("hr", null),
                             React.createElement("p", null, content));
}
```

This code is easy to read and understand: the curly braces are gone and the resulting code passes the incoming function parameters to `React.createElement()` in a natural way. Note the neat trick that we've done here with the `drawLine` parameter and the short-circuiting `&&` operator. If you call `sidebar()` with only three arguments, then `drawLine` defaults to `true`, and the fourth argument to the outer `createElement()` call is the `<hr/>` element. But if you pass `false` as the fourth argument to `sidebar()`, then the fourth argument to the outer `createElement()` call evaluates to `false`, and no `<hr/>` element is ever created. This use of the `&&` operator is a common idiom in JSX to conditionally include or exclude a child element depending on the value of some other expression. (This idiom works with React because React simply ignores children that are `false` or `null` and does not produce any output for them.)

When you use JavaScript expressions within JSX expressions, you are not limited to simple values like the string and boolean values in the preceding example. Any JavaScript value is allowed. In fact, it is quite common in React programming to use objects, arrays, and functions. Consider the following function, for example:

```
// Given an array of strings and a callback function return a JSX element
// representing an HTML <ul> list with an array of <li> elements as its child.
function list(items, callback) {
  return (
    <ul style={ {padding:10, border:"solid red 4px"} }>
      {items.map((item,index) => {
        <li onClick={() => callback(index)} key={index}>{item}</li>
      })}
    </ul>
  );
}
```

This function uses an object literal as the value of the `style` attribute on the `<ul>` element. (Note that double curly braces are required here.) The `<ul>` element has a single child, but the value of that child is an array. The child array is the array created by using the `map()` function on the input array to create an array of `<li>` elements.

(This works with React because the React library flattens the children of an element when it renders them. An element with one array child is the same as that element with each of those array elements as children.) Finally, note that each of the nested `<li>` elements has an `onClick` event handler attribute whose value is an arrow function. The JSX code compiles to the following pure JavaScript code (which I have formatted with Prettier):

```
function list(items, callback) {
  return React.createElement(
    "ul",
    { style: { padding: 10, border: "solid red 4px" } },
    items.map((item, index) =>
      React.createElement(
        "li",
        { onClick: () => callback(index), key: index },
        item
      )
    )
  );
}
```

One other use of object expressions in JSX is with the object spread operator (§6.10.4) to specify multiple attributes at once. Suppose that you find yourself writing a lot of JSX expressions that repeat a common set of attributes. You can simplify your expressions by defining the attributes as properties of an object and "spreading them into" your JSX elements:

```
let hebrew = { lang: "he", dir: "rtl" }; // Specify language and direction
let shalom = <span className="emphasis" {...hebrew}>שלום</span>;
```

Babel compiles this to use an `_extends()` function (omitted here) that combines that `className` attribute with the attributes contained in the `hebrew` object:

```
let shalom = React.createElement("span",
                                 _extends({className: "emphasis"}, hebrew),
                                 "\u05E9\u05DC\u05D5\u05DD");
```

Finally, there is one more important feature of JSX that we have not covered yet. As you've seen, all JSX elements begin with an identifier immediately after the opening angle bracket. If the first letter of this identifier is lowercase (as it has been in all of the examples here), then the identifier is passed to `createElement()` as a string. But if the first letter of the identifier is uppercase, then it is treated as an actual identifer, and it is the JavaScript value of that identifier that is passed as the first argument to `createElement()`. This means that the JSX expression `<Math/>` compiles to JavaScript code that passes the global Math object to `React.createElement()`.

For React, this ability to pass non-string values as the first argument to `createElement()` enables the creation of *components*. A component is a way of writing a simple

JSX expression (with an uppercase component name) that represents a more complex expression (using lowercase HTML tag names).

The simplest way to define a new component in React is to write a function that takes a "props object" as its argument and returns a JSX expression. A *props object* is simply a JavaScript object that represents attribute values, like the objects that are passed as the second argument to `createElement()`. Here, for example, is another take on our `sidebar()` function:

```
function Sidebar(props) {
  return (
    <div>
      <h1>{props.title}</h1>
      { props.drawLine && <hr/> }
      <p>{props.content}</p>
    </div>
  );
}
```

This new `Sidebar()` function is a lot like the earlier `sidebar()` function. But this one has a name that begins with a capital letter and takes a single object argument instead of separate arguments. This makes it a React component and means that it can be used in place of an HTML tag name in JSX expressions:

```
let sidebar = <Sidebar title="Something snappy" content="Something wise"/>;
```

This `<Sidebar/>` element compiles like this:

```
let sidebar = React.createElement(Sidebar, {
  title: "Something snappy",
  content: "Something wise"
});
```

It is a simple JSX expression, but when React renders it, it will pass the second argument (the Props object) to the first argument (the `Sidebar()` function) and will use the JSX expression returned by that function in place of the `<Sidebar>` expression.

17.8 Type Checking with Flow

Flow is a language extension that allows you to annotate your JavaScript code with type information, and a tool for checking your JavaScript code (both annotated and unannotated) for type errors. To use Flow, you start writing code using the Flow language extension to add type annotations. Then you run the Flow tool to analyze your code and report type errors. Once you have fixed the errors and are ready to run the code, you use Babel (perhaps automatically as part of the code-bundling process) to strip the Flow type annotations out of your code. (One of the nice things about the Flow language extension is that there isn't any new syntax that Flow has to compile or transform. You use the Flow language extension to add annotations to the code, and

all Babel has to do is to strip those annotations out to return your code to standard JavaScript.)

TypeScript Versus Flow

TypeScript is a very popular alternative to Flow. TypeScript is an extension of JavaScript that adds types as well as other language features. The TypeScript compiler "tsc" compiles TypeScript programs into JavaScript programs and in the process analyzes them and reports type errors in much the same the way that Flow does. tsc is not a Babel plugin: it is its own standalone compiler.

Simple type annotations in TypeScript are usually written identically to the same annotations in Flow. For more advanced typing, the syntax of the two extensions diverges, but the intent and value of the two extensions is the same. My goal in this section is to explain the benefits of type annotations and static code analysis. I'll be doing that with examples based on Flow, but everything demonstrated here can also be achieved with TypeScript with relatively simple syntax changes.

TypeScript was released in 2012, before ES6, when JavaScript did not have a `class` keyword or a `for/of` loop or modules or Promises. Flow is a narrow language extension that adds type annotations to JavaScript and nothing else. TypeScript, by contrast, was very much designed as a new language. As its name implies, adding types to JavaScript is the primary purpose of TypeScript, and it is the reason that people use it today. But types are not the only feature that TypeScript adds to JavaScript: the TypeScript language has `enum` and `namespace` keywords that simply do not exist in JavaScript. In 2020, TypeScript has better integration with IDEs and code editors (particularly VSCode, which, like TypeScript, is from Microsoft) than Flow does.

Ultimately, this is a book about JavaScript, and I'm covering Flow here instead of TypeScript because I don't want to take the focus off of JavaScript. But everything you learn here about adding types to JavaScript will be helpful to you if you decide to adopt TypeScript for your projects.

Using Flow requires commitment, but I have found that for medium and large projects, the extra effort is worth it. It takes extra time to add type annotations to your code, to run Flow every time you edit the code, and to fix the type errors it reports. But in return Flow will enforce good coding discipline and will not allow you to cut corners that can lead to bugs. When I have worked on projects that use Flow, I have been impressed by the number of errors it found in my own code. Being able to fix those issues before they became bugs is a great feeling and gives me extra confidence that my code is correct.

When I first started using Flow, I found that it was sometimes difficult to understand why it was complaining about my code. With some practice, though, I came to understand its error messages and found that it was usually easy to make minor

changes to my code to make it safer and to satisfy Flow.[1] I do not recommend using Flow if you still feel like you are learning JavaScript itself. But once you are confident with the language, adding Flow to your JavaScript projects will push you to take your programming skills to the next level. And this, really, is why I'm dedicating the last section of this book to a Flow tutorial: because learning about JavaScript type systems offers a glimpse of another level, or another style, of programming.

This section is a tutorial, and it does not attempt to cover Flow comprehensively. If you decide to try Flow, you will almost certainly end up spending time reading the documentation at *https://flow.org*. On the other hand, you do not need to master the Flow type system before you can start making practical use of it in your projects: the simple uses of Flow described here will take you a long way.

17.8.1 Installing and Running Flow

Like the other tools described in this chapter, you can install the Flow type-checking tool using a package manager, with a command like `npm install -g flow-bin` or `npm install --save-dev flow-bin`. If you install the tool globally with `-g`, then you can run it with `flow`. And if you install it locally in your project with `--save-dev`, then you can run it with `npx flow`. Before using Flow to do type checking, the first time run it as `flow --init` in the root directory of your project to create a `.flowconfig` configuration file. You may never need to add anything to this file, but Flow needs it to know where your project root is.

When you run Flow, it will find all the JavaScript source code in your project, but it will only report type errors for the files that have "opted in" to type checking by adding a `// @flow` comment at the top of the file. This opt-in behavior is important because it means that you can adopt Flow for existing projects and then begin to convert your code one file at a time, without being bothered by errors and warnings on files that have not yet been converted.

Flow may be able to find errors in your code even if all you do is opt in with a `// @flow` comment. Even if you do not use the Flow language extension and add no type annotations to your code, the Flow type checker tool can still make inferences about the values in your program and alert you when you use them inconsistently.

Consider the following Flow error message:

```
Error ------------------------------------------------------ variableReassignment.js:6:3

Cannot assign 1 to i.r because:
```

1 If you have programmed with Java, you may have experienced something like this the first time you wrote a generic API that used a type parameter. I found the learning process for Flow to be remarkably similar to what I went through in 2004 when generics were added to Java.

```
• property r is missing in number [1].

    2| let i = { r: 0, i: 1 };    // The complex number 0+1i
[1] 3| for(i = 0; i < 10; i++) {  // Oops! The loop variable overwrites i
    4|     console.log(i);
    5| }
    6| i.r = 1;                  // Flow detects the error here
```

In this case, we declare the variable `i` and assign an object to it. Then we use `i` again as a loop variable, overwriting the object. Flow notices this and flags an error when we try to use `i` as if it still held an object. (A simple fix would be to write `for(let i = 0;` making the loop variable local to the loop.)

Here is another error that Flow detects even without type annotations:

```
Error ····· size.js:3:14

Cannot get x.length because property length is missing in Number [1].

    1| // @flow
    2| function size(x) {
    3|     return x.length;
    4| }
[1] 5| let s = size(1000);
```

Flow sees that the `size()` function takes a single argument. It doesn't know the type of that argument, but it can see that the argument is expected to have a `length` property. When it sees this `size()` function being called with a numeric argument, it correctly flags this as an error because numbers do not have `length` properties.

17.8.2 Using Type Annotations

When you declare a JavaScript variable, you can add a Flow type annotation to it by following the variable name with a colon and the type:

```
let message: string = "Hello world";
let flag: boolean = false;
let n: number = 42;
```

Flow would know the types of these variables even if you did not annotate them: it can see what values you assign to each variable, and it keeps track of that. If you add type annotations, however, Flow knows both the type of the variable and that you have expressed the intent that the variable should always be of that type. So if you use the type annotation, Flow will flag an error if you ever assign a value of a different type to that variable. Type annotations for variables are also particularly useful if you tend to declare all your variables up at the top of a function before they are used.

Type annotations for function arguments are like annotations for variables: follow the name of the function argument with a colon and the type name. When annotating a function, you typically also add an annotation for the return type of the function.

This goes between the close parenthesis and the open curly brace of the function body. Functions that return nothing use the Flow type `void`.

In the preceding example we defined a `size()` function that expected an argument with a `length` property. Here's how we could change that function to explicitly specify that it expects a string argument and returns a number. Note, Flow now flags an error if we pass an array to the function, even though the function would work in that case:

```
Error ------------------------------------------------------ size2.js:5:18

Cannot call size with array literal bound to s because array literal [1]
is incompatible with string [2].

 [2] 2| function size(s: string): number {
     3|     return s.length;
     4| }
 [1] 5| console.log(size([1,2,3]));
```

Using type annotations with arrow functions is also possible, though it can turn this normally succinct syntax into something more verbose:

```
const size = (s: string): number => s.length;
```

An important thing to understand about Flow is that the JavaScript value `null` has the Flow type `null` and the JavaScript value `undefined` has the Flow type `void`. But neither of these values is a member of any other type (unless you explicitly add it). If you declare a function parameter to be a string, then it must be a string, and it is an error to pass `null` or to pass `undefined` or to omit the argument (which is basically the same thing as passing `undefined`):

```
Error ------------------------------------------------------ size3.js:3:18

Cannot call size with null bound to s because null [1] is incompatible
with string [2].

     1| // @flow
 [2] 2| const size = (s: string): number => s.length;
 [1] 3| console.log(size(null));
```

If you want to allow `null` and `undefined` as legal values for a variable or function argument, simply prefix the type with a question mark. For example, use `?string` or `?number` instead of `string` or `number`. If we change our `size()` function to expect an argument of type `?string`, then Flow doesn't complain when we pass `null` to the function. But it now has something else to complain about:

```
Error ------------------------------------------------------ size4.js:3:14

Cannot get s.length because property length is missing in null or
undefined [1].

     1| // @flow
```

```
[1] 2| function size(s: ?string): number {
    3|     return s.length;
    4| }
    5| console.log(size(null));
```

What Flow is telling us here is that it is not safe to write `s.length` because, at this place in our code, `s` might be `null` or `undefined`, and those values do not have `length` properties. This is where Flow makes sure we do not cut any corners. If a value might be `null`, Flow will insist that we check for that case before we do anything that depends on the value not being `null`.

In this case, we can fix the issue by changing the body of the function as follows:

```
function size(s: ?string): number {
    // At this point in the code, s could be a string or null or undefined.
    if (s === null || s === undefined) {
        // In this block, Flow knows that s is null or undefined.
        return -1;
    } else {
        // And in this block, Flow knows that s is a string.
        return s.length;
    }
}
```

When the function is first called, the parameter can have more than one type. But by adding type-checking code, we create a block within the code where Flow knows for sure that the parameter is a string. When we use `s.length` within that block, Flow does not complain. Note that Flow does not require you to write verbose code like this. Flow would also be satisfied if we just replaced the body of the `size()` function with `return s ? s.length : -1;`.

Flow syntax allows a question mark before any type specification to indicate that, in addition to the specified type, `null` and `undefined` are allowed as well. Question marks can also appear after a parameter name to indicate that the parameter itself is optional. So if we changed the declaration of the parameter `s` from `s: ?string` to `s? : string`, that would mean it is OK to call `size()` with no arguments (or with the value `undefined`, which is the same as omitting it), but that if we do call it with a parameter other than `undefined`, then that parameter must be a string. In this case, `null` is not a legal value.

So far, we've discussed primitive types `string`, `number`, `boolean`, `null`, and `void` and have demonstrated how you can use use them with variable declarations, function parameters, and function return values. The subsections that follow describe some more complex types supported by Flow.

17.8.3 Class Types

In addition to the primitive types that Flow knows about, it also knows about all of JavaScript's built-in classes and allows you to use class name as types. The following function, for example, uses type annotations to indicate that it should be invoked with one Date object and one RegExp object:

```
// @flow
// Return true if the ISO representation of the specified date
// matches the specified pattern, or false otherwise.
// E.g: const isTodayChristmas = dateMatches(new Date(), /^\d{4}-12-25T/);
export function dateMatches(d: Date, p: RegExp): boolean {
    return p.test(d.toISOString());
}
```

If you define your own classes with the `class` keyword, those classes automatically become valid Flow types. In order to make this work, however, Flow does require you to use type annotations in the class. In particular, each property of the class must have its type declared. Here is a simple complex number class that demonstrates this:

```
// @flow
export default class Complex {
    // Flow requires an extended class syntax that includes type annotations
    // for each of the properties used by the class.
    i: number;
    r: number;
    static i: Complex;

    constructor(r: number, i:number) {
        // Any properties initialized by the constructor must have Flow type
        // annotations above.
        this.r = r;
        this.i = i;
    }

    add(that: Complex) {
        return new Complex(this.r + that.r, this.i + that.i);
    }
}

// This assignment would not be allowed by Flow if there was not a
// type annotation for i inside the class.
Complex.i = new Complex(0,1);
```

17.8.4 Object Types

The Flow type to describe an object looks a lot like an object literal, except that property values are replaced by property types. Here, for example, is a function that expects an object with numeric `x` and `y` properties:

```
// @flow
// Given an object with numeric x and y properties, return the
// distance from the origin to the point (x,y) as a number.
export default function distance(point: {x:number, y:number}): number {
    return Math.hypot(point.x, point.y);
}
```

In this code, the text `{x:number, y:number}` is a Flow type, just like `string` or `Date` is. As with any type, you can add a question mark at the front to indicate that `null` and `undefined` should also be allowed.

Within an object type, you can follow any of the property names with a question mark to indicate that that property is optional and may be omitted. For example, you might write the type for an object that represents a 2D or 3D point like this:

```
{x: number, y: number, z?: number}
```

If a property is not marked as optional in an object type, then it is required, and Flow will report an error if an appropriate property is not present in the actual value. Normally, however, Flow tolerates extra properties. If you were to pass an object that had a `w` property to the `distance()` function above, Flow would not complain.

If you want Flow to strictly enforce that an object does not have properties other than those explicitly declared in its type, you can declare an *exact object type* by adding vertical bars to the curly braces:

```
{| x: number, y: number |}
```

JavaScript's objects are sometimes used as dictionaries or string-to-value maps. When used like this, the property names are not known in advance and cannot be declared in a Flow type. If you use objects this way, you can still use Flow to describe the data structure. Suppose that you have an object where the properties are the names of the world's major cities and the values of those properties are objects that specify the geographical location of those cities. You might declare this data structure like this:

```
// @flow
const cityLocations : {[string]: {longitude:number, latitude:number}} = {
    "Seattle": { longitude: 47.6062, latitude: -122.3321 },
    // TODO: if there are any other important cities, add them here.
};
export default cityLocations;
```

17.8.5 Type Aliases

Objects can have many properties, and the Flow type that describes such an object will be long and difficult to type. And even relatively short object types can be confusing because they look so much like object literals. Once we get beyond simple types like `number` and `?string`, it is often useful to be able to define names for our Flow types. And in fact, Flow uses the `type` keyword to do exactly that. Follow the `type`

keyword with an identifier, an equals sign, and a Flow type. Once you've done that, the identifier will be an alias for the type. Here, for example, is how we could rewrite the `distance()` function from the previous section with an explicitly defined `Point` type:

```
// @flow
export type Point = {
    x: number,
    y: number
};

// Given a Point object return its distance from the origin
export default function distance(point: Point): number {
    return Math.hypot(point.x, point.y);
}
```

Note that this code exports the `distance()` function and also exports the `Point` type. Other modules can use `import type Point from './distance.js'` if they want to use that type definition. Keep in mind, though, that `import type` is a Flow language extension and not a real JavaScript import directive. Type imports and exports are used by the Flow type checker, but like all other Flow language extensions, they are stripped out of the code before it ever runs.

Finally, it is worth noting that instead of defining a name for a Flow object type that represents a point, it would probably be simpler and cleaner to just define a Point class and use that class as the type.

17.8.6 Array Types

The Flow type to describe an array is a compound type that also includes the type of the array elements. Here, for example, is a function that expects an array of numbers, and the error that Flow reports if you try to call the function with an array that has non-numeric elements:

```
Error ------------------------------------------------------ average.js:8:16

Cannot call average with array literal bound to data because string [1]
is incompatible with number [2] in array element.

 [2]  2| function average(data: Array<number>) {
      3|     let sum = 0;
      4|     for(let x of data) sum += x;
      5|     return sum/data.length;
      6| }
      7|
 [1]  8| average([1, 2, "three"]);
```

The Flow type for an array is `Array` followed by the element type in angle brackets. You can also express an array type by following the element type with open and close

square brackets. So in this example we could have written `number[]` instead of `Array<number>`. I prefer the angle bracket notation because, as we'll see, there are other Flow types that use this angle-bracket syntax.

The Array type syntax shown works for arrays with an arbitrary number of elements, all of which have the same type. Flow has a different syntax for describing the type of a *tuple*: an array with a fixed number of elements, each of which may have a different type. To express the type of a tuple, simply write the type of each of its elements, separate them with commas, and enclose them all in square brackets.

A function that returns an HTTP status code and message might look like this, for example:

```
function getStatus():[number, string] {
    return [getStatusCode(), getStatusMessage()];
}
```

Functions that return tuples are awkward to work with unless you use destructuring assignment:

```
let [code, message] = getStatus();
```

Destructuring assignment, plus Flow's type-aliasing capabilities, make tuples easy enough to work with that you might consider them as an alternative to classes for simple datatypes:

```
// @flow
export type Color = [number, number, number, number];  // [r, g, b, opacity]

function gray(level: number): Color {
    return [level, level, level, 1];
}

function fade([r,g,b,a]: Color, factor: number): Color {
    return [r, g, b, a/factor];
}

let [r, g, b, a] = fade(gray(75), 3);
```

Now that we have a way to express the type of an array, let's return to the `size()` function from earlier and modify it to expect an array argument instead of a string argument. We want the function to be able to accept an array of any length, so a tuple type is not appropriate. But we don't want to restrict our function to working only for arrays where all the elements have the same type. The solution is the type `Array<mixed>`:

```
// @flow
function size(s: Array<mixed>): number {
    return s.length;
}
console.log(size([1,true,"three"]));
```

The element type `mixed` indicates that the elements of the array can be of any type. If our function actually indexed the array and attempted to use any of those elements, Flow would insist that we use `typeof` checks or other tests to determine the type of the element before performing any unsafe operation on it. (If you are willing to give up on type checking, you can also use `any` instead of `mixed`: it allows you to do whatever you want with the values of the array without ensuring that the values are of the type you expect.)

17.8.7 Other Parameterized Types

We've seen that when you annotate a value as an `Array`, Flow requires you to also specify the type of the array elements inside angle brackets. This additional type is known as a *type parameter*, and Array is not the only JavaScript class that is parameterized.

JavaScript's Set class is a collection of elements, like an array is, and you can't use `Set` as a type by itself, but you have to include a type parameter within angle brackets to specify the type of the values contained in the set. (Though you can use `mixed` or `any` if the set may contain values of multiple types.) Here's an example:

```
// @flow
// Return a set of numbers with members that are exactly twice those
// of the input set of numbers.
function double(s: Set<number>): Set<number> {
    let doubled: Set<number> = new Set();
    for(let n of s) doubled.add(n * 2);
    return doubled;
}
console.log(double(new Set([1,2,3])));  // Prints "Set {2, 4, 6}"
```

Map is another parameterized type. In this case, there are two type parameters that must be specified; the type of the keys and the types of the values:

```
// @flow
import type { Color } from "./Color.js";

let colorNames: Map<string, Color> = new Map([
    ["red", [1, 0, 0, 1]],
    ["green", [0, 1, 0, 1]],
    ["blue", [0, 0, 1, 1]]
]);
```

Flow lets you define type parameters for your own classes as well. The following code defines a Result class but parameterizes that class with an Error type and a Value type. We use placeholders `E` and `V` in the code to represent these type parameters. When the user of this class declares a variable of type Result, they will specify the actual types to substitute for `E` and `V`. The variable declaration might look like this:

```
let result: Result<TypeError, Set<string>>;
```

And here is how the parameterized class is defined:

```
// @flow
// This class represents the result of an operation that can either
// throw an error of type E or a value of type V.
export class Result<E, V> {
    error: ?E;
    value: ?V;

    constructor(error: ?E, value: ?V) {
        this.error = error;
        this.value = value;
    }

    threw(): ?E { return this.error; }
    returned(): ?V { return this.value; }

    get():V {
        if (this.error) {
            throw this.error;
        } else if (this.value === null || this.value === undefined) {
            throw new TypeError("Error and value must not both be null");
        } else {
            return this.value;
        }
    }

}
```

And you can even define type parameters for functions:

```
// @flow
// Combine the elements of two arrays into an array of pairs
function zip<A,B>(a:Array<A>, b:Array<B>): Array<[?A,?B]> {
    let result:Array<[?A,?B]> = [];
    let len = Math.max(a.length, b.length);
    for(let i = 0; i < len; i++) {
        result.push([a[i], b[i]]);
    }
    return result;
}

// Create the array [[1,'a'], [2,'b'], [3,'c'], [4,undefined]]
let pairs: Array<[?number,?string]> = zip([1,2,3,4], ['a','b','c'])
```

17.8.8 Read-Only Types

Flow defines some special parameterized "utility types" that have names beginning with `$`. Most of these types have advanced use cases that we are not going to cover here. But two of them are quite useful in practice. If you have an object type T and want to make a read-only version of that type, just write `$ReadOnly<T>`. Similarly, you can write `$ReadOnlyArray<T>` to describe a read-only array with elements of type T.

The reason to use these types is not because they can offer any guarantee that an object or array can't be modified (see `Object.freeze()` in §14.2 if you want true read-only objects) but because it allows you to catch bugs caused by unintentional modifications. If you write a function that takes an object or array argument and does not change any of the object's properties or the array's elements, then you can annotate the function parameter with one of Flow's read-only types. If you do this, then Flow will report an error if you forget and accidentally modify the input value. Here are two examples:

```
// @flow
type Point = {x:number, y:number};

// This function takes a Point object but promises not to modify it
function distance(p: $ReadOnly<Point>): number {
    return Math.hypot(p.x, p.y);
}

let p: Point = {x:3, y:4};
distance(p)  // => 5

// This function takes an array of numbers that it will not modify
function average(data: $ReadOnlyArray<number>): number {
    let sum = 0;
    for(let i = 0; i < data.length; i++) sum += data[i];
    return sum/data.length;
}

let data: Array<number> = [1,2,3,4,5];
average(data) // => 3
```

17.8.9 Function Types

We have seen how to add type annotations to specify the types of a function's parameters and its return type. But when one of the parameters of a function is itself a function, we need to be able to specify the type of that function parameter.

To express the type of a function with Flow, write the types of each parameter, separate them with commas, enclose them in parentheses, and then follow that with an arrow and type return type of the function.

Here is an example function that expects to be passed a callback function. Notice how we defined a type alias for the type of the callback function:

```
// @flow
// The type of the callback function used in fetchText() below
export type FetchTextCallback = (?Error, ?number, ?string) => void;

export default function fetchText(url: string, callback: FetchTextCallback) {
    let status = null;
    fetch(url)
```

```
        .then(response => {
            status = response.status;
            return response.text()
        })
        .then(body => {
            callback(null, status, body);
        })
        .catch(error => {
            callback(error, status, null);
        });
}
```

17.8.10 Union Types

Let's return one more time to the `size()` function. It doesn't really make sense to have a function that does nothing other than return the length of an array. Arrays have a perfectly good `length` property for that. But `size()` might be useful if it could take any kind of collection object (an array or a Set or a Map) and return the number of elements in the collection. In regular untyped JavaScript it would be easy to write a `size()` function like that. With Flow, we need a way to express a type that allows arrays, Sets, and Maps, but doesn't allow values of any other type.

Flow calls types like this *Union types* and allows you to express them by simply listing the desired types and separating them with vertical bar characters:

```
// @flow
function size(collection: Array<mixed>|Set<mixed>|Map<mixed,mixed>): number {
    if (Array.isArray(collection)) {
        return collection.length;
    } else {
        return collection.size;
    }
}
size([1,true,"three"]) + size(new Set([true,false])) // => 5
```

Union types can be read using the word "or"—"an array or a Set or a Map"—so the fact that this Flow syntax uses the same vertical bar character as JavaScript's OR operators is intentional.

We saw earlier that putting a question mark before a type allows `null` and `undefined` values. And now you can see that a `?` prefix is simply a shortcut for adding a `|null|void` suffix to a type.

In general, when you annotate a value with a Union type, Flow will not allow you to use that value until you've done enough tests to figure out what the type of the actual value is. In the `size()` example we just looked at, we need to explicitly check whether the argument is an array before we try to access the `length` property of the argument. Note that we do not have to distinguish a Set argument from a Map argument,

however: both of those classes define a `size` property, so the code in the `else` clause is safe as long as the argument is not an array.

17.8.11 Enumerated Types and Discriminated Unions

Flow allows you to use primitive literals as types that consist of that one single value. If you write `let x:3;`, then Flow will not allow you to assign any value to that variable other than 3. It is not often useful to define types that have only a single member, but a union of literal types can be useful. You can probably imagine a use for types like these, for example:

```
type Answer = "yes" | "no";
type Digit = 0|1|2|3|4|5|6|7|8|9;
```

If you use types made up of literals, you need to understand that only literal values are allowed:

```
let a: Answer = "Yes".toLowerCase(); // Error: can't assign string to Answer
let d: Digit = 3+4;                  // Error: can't assign number to Digit
```

When Flow checks your types, it does not actually do the calculations: it just checks the types of the calculations. Flow knows that `toLowerCase()` returns a string and that the `+` operator on numbers returns a number. Even though we know that both of these calculations return values that are within the type, Flow cannot know that and flags errors on both of these lines.

A union type of literal types like `Answer` and `Digit` is an example of an *enumerated type*, or *enum*. A canonical use case for enum types is to represent the suits of playing cards:

```
type Suit = "Clubs" | "Diamonds" | "Hearts" | "Spades";
```

A more relevant example might be HTTP status codes:

```
type HTTPStatus =
    | 200    // OK
    | 304    // Not Modified
    | 403    // Forbidden
    | 404;   // Not Found
```

One of the pieces of advice that new programmers often hear is to avoid using literals in their code and to instead define symbolic constants to represent those values. One practical reason for this is to avoid the problem of typos: if you misspell a string literal like "Diamonds" JavaScript may never complain but your code may not work right. If you mistype an identifier, on the other hand, JavaScript is likely to throw an error that you'll notice. With Flow, this advice does not always apply. If you annotate a variable with the type Suit, and then try to assign a misspelled suit to it, Flow will alert you to the error.

Another important use for literal types is the creation of *discriminated unions*. When you work with union types (made up of actually different types, not of literals), you typically have to write code to discriminate among the possible types. In the previous section, we wrote a function that could take an array or a Set or a Map as its argument and had to write code to discriminate array input from Set or Map input. If you want to create a union of Object types, you can make these types easy to discriminate by using a literal type within each of the individual Object types.

An example will make this clear. Suppose you're using a worker thread in Node (§16.11) and are using `postMessage()` and "message" events for sending object-based messages between the main thread and the worker thread. There are multiple types of messages that the worker might want to send to the main thread, but we'd like to write a Flow Union type that describes all possible messages. Consider this code:

```
// @flow
// The worker sends a message of this type when it is done
// reticulating the splines we sent it.
export type ResultMessage = {
    messageType: "result",
    result: Array<ReticulatedSpline>, // Assume this type is defined elsewhere.
};

// The worker sends a message of this type if its code failed with an exception.
export type ErrorMessage = {
    messageType: "error",
    error: Error,
};

// The worker sends a message of this type to report usage statistics.
export type StatisticsMessage = {
    messageType: "stats",
    splinesReticulated: number,
    splinesPerSecond: number
};

// When we receive a message from the worker it will be a WorkerMessage.
export type WorkerMessage = ResultMessage | ErrorMessage | StatisticsMessage;

// The main thread will have an event handler function that is passed
// a WorkerMessage. But because we've carefully defined each of the
// message types to have a messageType property with a literal type,
// the event handler can easily discriminate among the possible messages:
function handleMessageFromReticulator(message: WorkerMessage) {
    if (message.messageType === "result") {
        // Only ResultMessage has a messageType property with this value
        // so Flow knows that it is safe to use message.result here.
        // And Flow will complain if you try to use any other property.
        console.log(message.result);
    } else if (message.messageType === "error") {
        // Only ErrorMessage has a messageType property with value "error"
```

```
            // so knows that it is safe to use message.error here.
            throw message.error;
        } else if (message.messageType === "stats") {
            // Only StatisticsMessage has a messageType property with value "stats"
            // so knows that it is safe to use message.splinesPerSecond here.
            console.info(message.splinesPerSecond);
        }
    }
```

17.9 Summary

JavaScript is the most-used programming language in the world today. It is a living language—one that continues to evolve and improve—surrounded by a flourishing ecosystem of libraries, tools, and extensions. This chapter introduced some of those tools and extensions, but there are many more to learn about. The JavaScript ecosystem flourishes because the JavaScript developer community is active and vibrant, full of peers who share their knowledge through blog posts, videos, and conference presentations. As you close this book and go forth to join this community, you will find no shortage of information sources to keep you engaged with and learning about JavaScript.

Best wishes, David Flanagan, March 2020

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

0

P

Q

R

S

T

U

V

W

X

Y

Z

About the Author

David Flanagan has been programming with and writing about JavaScript since 1995. He lives with his wife and children in the Pacific Northwest between the cities of Seattle, Washington, and Vancouver, British Columbia. David has a degree in computer science and engineering from the Massachusetts Institute of Technology and works as a software engineer at VMware.

Colophon

The animal on the cover of *JavaScript: The Definitive Guide*, Seventh Edition, is a Javan rhinoceros (*Rhinoceros sondaicus*). All five species of rhinoceros are distinguished by their large size, thick armor-like skin, three-toed feet, and single or double snout horn. The Javan rhinoceros resembles the related Indian rhinoceros, and as with that species, the males have a single horn. However, Javan rhinos are smaller and have unique skin textures. Though found today only in Indonesia, Javan rhinos once ranged throughout Southeastern Asia. They live in rainforest habitats, where they graze on abundant leaves and grasses and hide from insect pests such as blood-sucking flies by standing up to their snouts in water or mud.

The Javan rhino averages about 6 feet in height and can be up to 10 feet in length, with adults weighing up to 3,000 pounds. Like the Indian rhinoceros its gray skin seems to be separated into "plates," some of them textured. The natural lifespan of a Javan rhino is estimated at 45–50 years. Females give birth every 3–5 years, after a gestation period of 16 months. Calves weigh about 100 pounds when born, and stay with their protective mothers for up to 2 years.

Rhinoceros are generally a somewhat plentiful animal, being adaptable to a range of habitats and at adulthood having no natural predators. However, humans have hunted them nearly to extinction. Folklore holds that the horn of the rhinoceros possesses magical and aphrodisiac powers, and because of this, rhinos are a prime target for poachers. The Javan rhino population is the most precarious: as of 2020, the 70 or so remaining animals of this species live, under guard, in Ujung Kulon National Park, in Java, Indonesia. This strategy seems to be helping ensure the survival of these rhinos for the time being, as a 1967 census counted only 25.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The color illustration on the cover is by Karen Montgomery, based on a black-and-white engraving from Dover Animals. The cover fonts are Gilroy and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.