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.

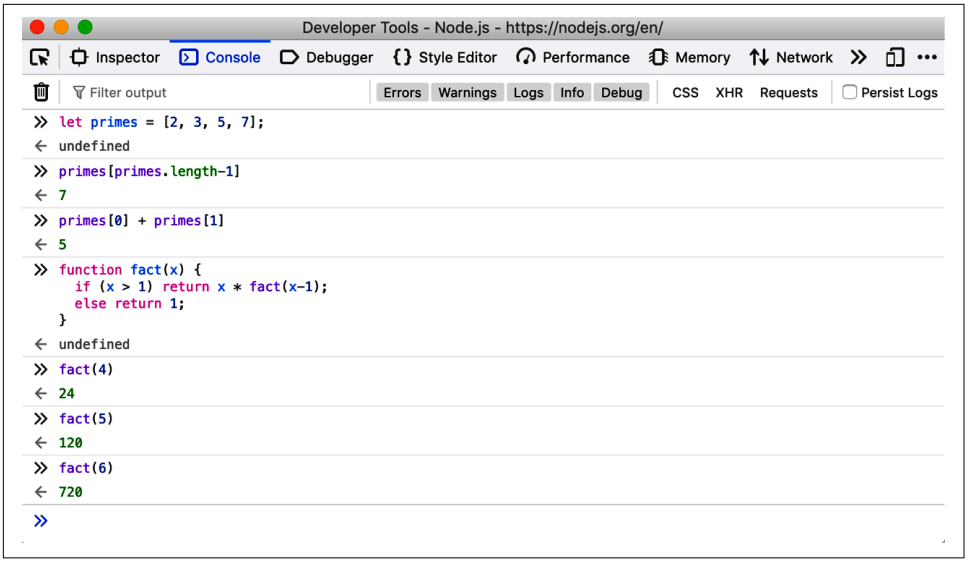

Figure 1-1. The JavaScript console in Firefox's Developer Tools

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.