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. 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="clo
    let now = new Date();                         // Get current time
    clock.textContent = now.toLocaleTimeString(); // Display time in the cloc
}
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> elemen
        s.onload = () => { resolve(); };          // Resolve promise when loa
        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.

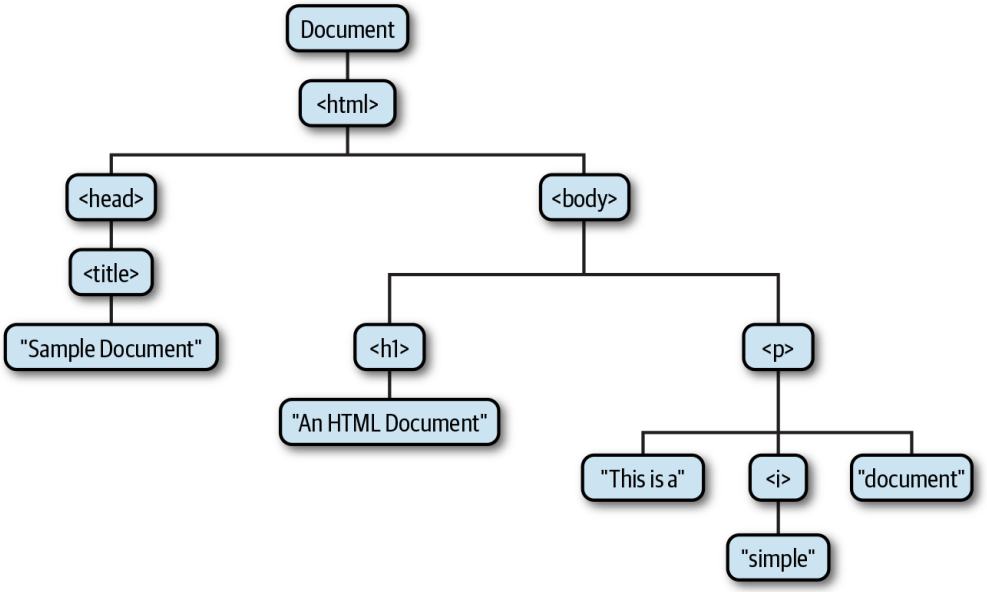

Figure 15-1. The tree representation of an HTML document

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>` ele-

ment) 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 sec-

tion 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.

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"). 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 elsewher
    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 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 <i
h2 ~ p                  // Any <p> that follows an <h2> and is a sibling of i
```

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"> elemen
```

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 sty
    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 low-

ercase 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.

WARNING

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” 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.

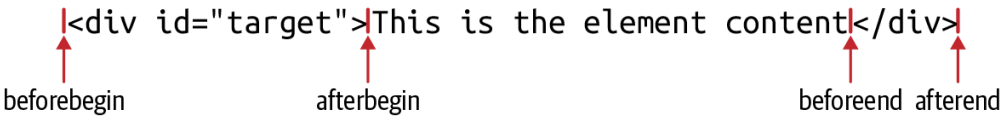

Figure 15-2. Insertion points for insertAdjacentHTML()

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".

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 stylesh
    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.

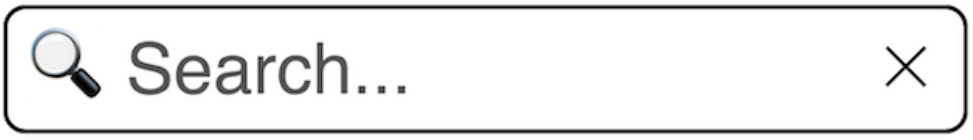

Figure 15-3. A search box user interface component

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.

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>
```

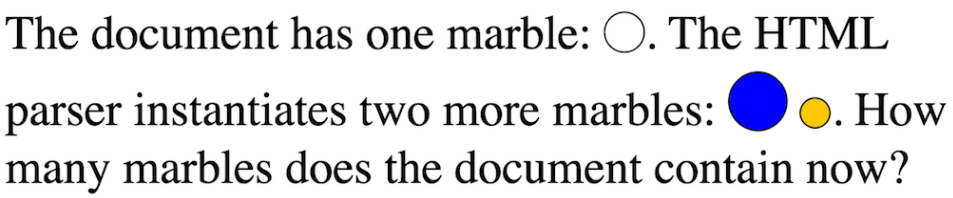

Figure 15-4. An inline circle custom element

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
 * 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 magnifyin
 * 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 elemen
 * 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 retur
    // 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 require
  // 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 periodi-

cally 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.

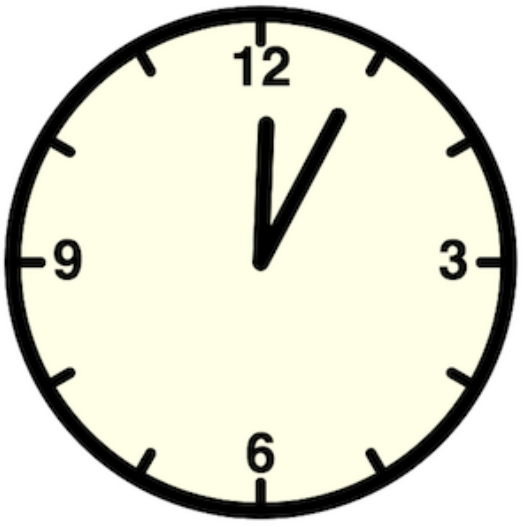

Figure 15-5. A scripted SVG analog clock

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 tim
    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."

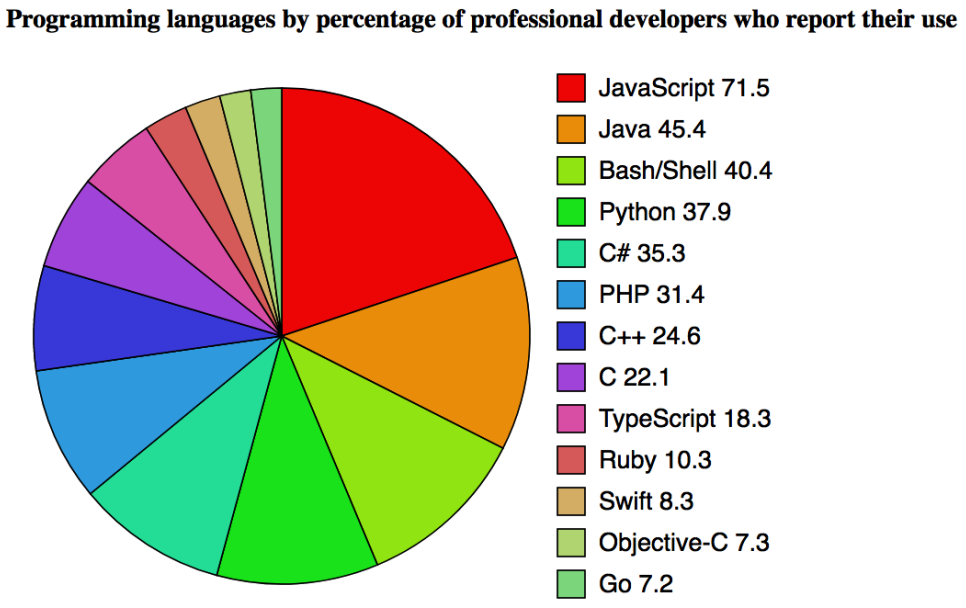

Figure 15-6. An SVG pie chart built with JavaScript (data from Stack Overflow's 2018 Developer Survey of Most Popular Technologies)

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 sli
    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.

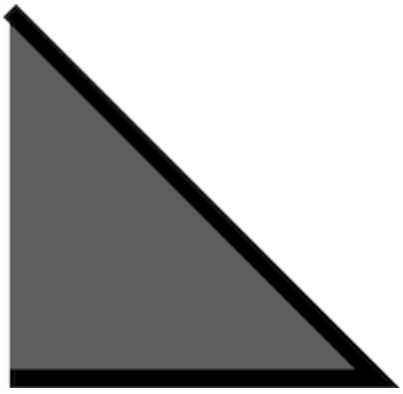

Figure 15-7. A simple path, filled and stroked

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.

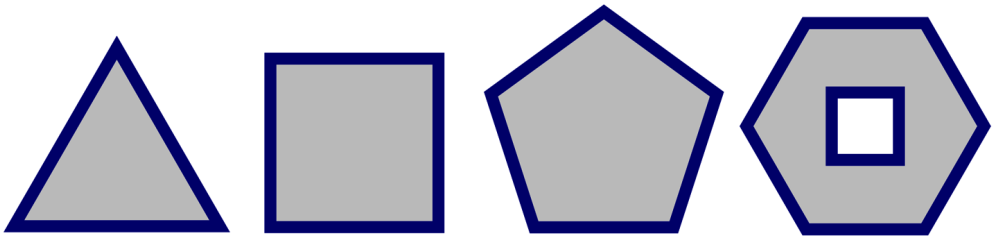

Figure 15-8. Regular polygons

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 vert
             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 hexag

// 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`.

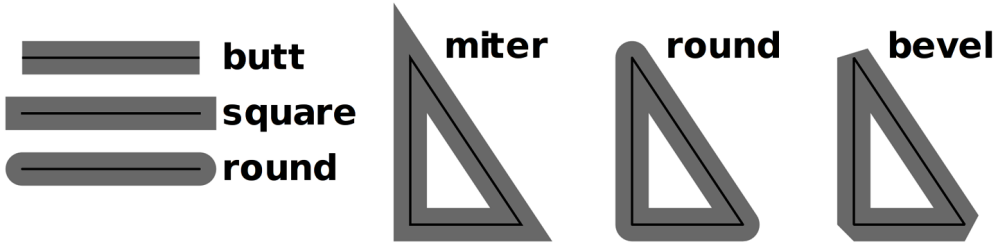

Figure 15-9. The lineCap and lineJoin attributes

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"). 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 shad-

ows. 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.

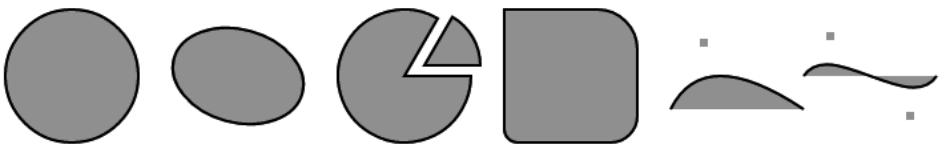

Figure 15-10. Curved paths in a canvas

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.

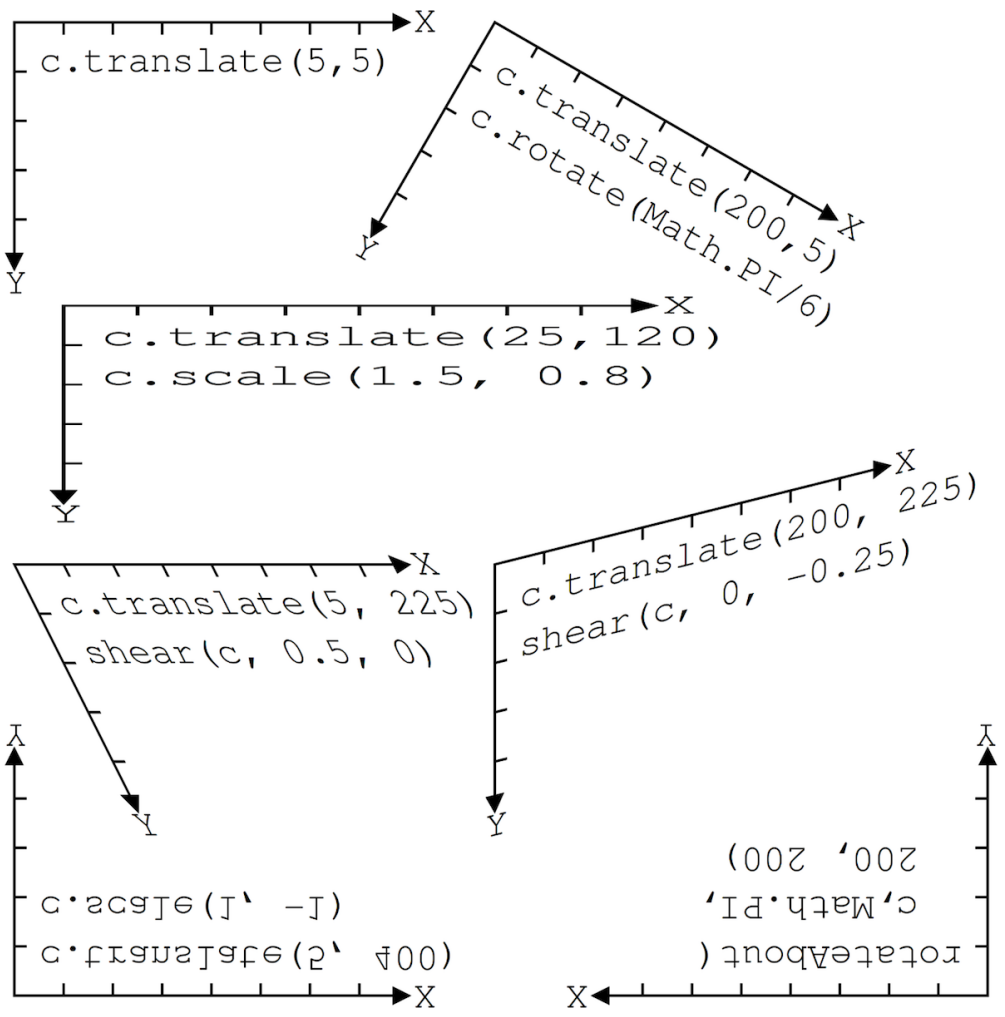

Figure 15-11. Coordinate system transformations

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.

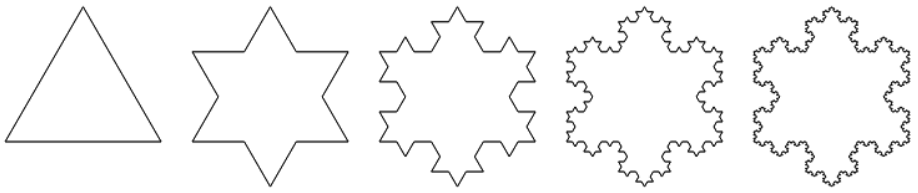

Figure 15-12. Koch snowflakes

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,
        // 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.

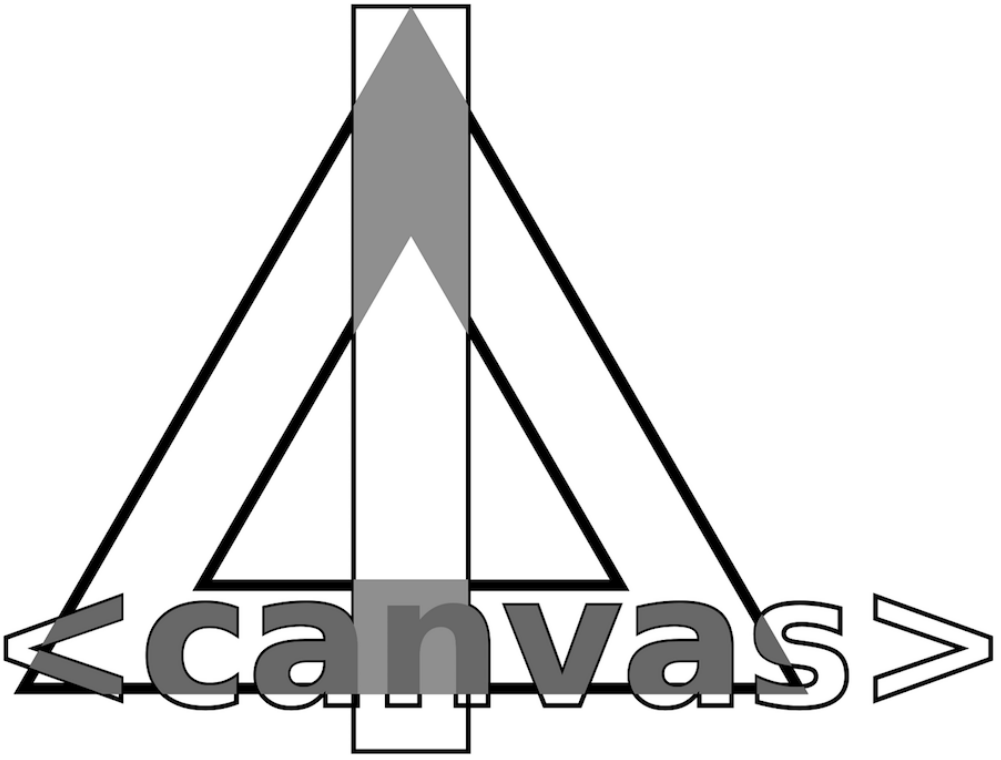

Figure 15-13. Unclipped strokes and clipped fills

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 regi
```

```
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 sm
    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 coul
    // 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 an
    // top-to-bottom. Each pixel occupies 4 consecutive bytes in R,G,B,A orde
    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 ro
        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 canva
    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()` construc-

tor 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 documen
```

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.

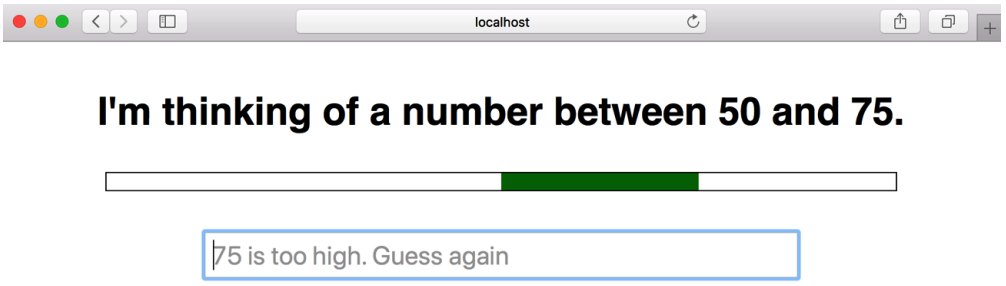

Figure 15-15. A number-guessing game

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</butto
<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 jus
// a plain JavaScript object that gets saved, not an instance of GameStat
// 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
    let range = document.querySelector("#range");     // Display guess ra
    let input = document.querySelector("#input");     // Guess input fiel
    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 typ
            );
        }
    })
    .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 erro
        // 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 processChun
 * 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 fa
    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 passe
                if (processed) {
                    body += processed;
```

```
                }
            } else {                                // Otherwise, convert bytes
                body += decoder.decode(value, {stream: true}); // to text.
            }

            if (reportProgress) {                   // If a progress callback w
                bytesRead += value.length;          // passed, then call it
                reportProgress(bytesRead, bytesRead / expectedBytes);
            }
        }
        if (done) {                                 // If this is the last chun
            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 con-

nection 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 error
    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 connecti
    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") 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", 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 valu
    }
    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 life-

time, 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"), 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 databas
    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 databas
        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 t
// the callback function.
//
// Our zip code database includes one object store that holds objects like th
//
//   {
//     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"), 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 w
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.

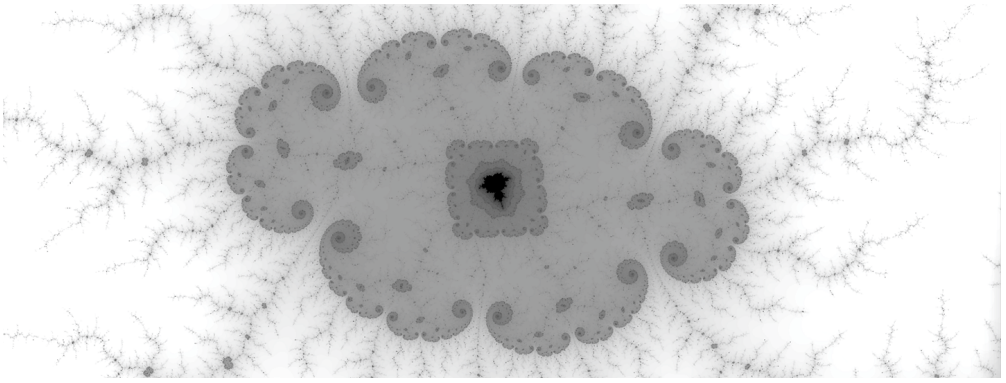

Figure 15-16. A portion of the Mandelbrot set

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 th
    //    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 dimension
    //  - 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
    // four individual bytes. We'll store the number of iterations for each
    // pixel in this iterations array. (The iterations will be transformed in
    // 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 iterati
                    }
                    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 pix
                if (n > max) max = n;       // Track the maximum number we've see
                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 Worker
 */
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, simpl
 * 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 workin
        this.workQueue = [];         // Work not currently being processed
        this.workerMap = new Map();  // Map workers to resolve and reject fun
```

```
        // 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 se
 * 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, a
 * 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 rea
 * 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 param
        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 nul
        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 WorkerP
            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 frequent
            this.colorTable = null;     // For converting raw data to pixel value

            // 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 simp
    // copy the properties of that object into the state object. If the optio
    // 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 even
        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 render
        // 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 skippin
        // 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 iteration
            // 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 long
```

```
            // the right size, then allocate a new one.
            if (!this.colorTable || this.colorTable.length !== maxIterations+
                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 h
            // 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 sa
                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 a
                // 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 m
            // 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 no
            if (this.wantsRerender) {
                this.wantsRerender = false;
                this.render();
            }
        });
    }

// If the user resizes the window, this function will be called repeatedl
// 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 coordinate
        // 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()-t
                // 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 cam-

era. 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.

[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.

[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.

[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.

[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.