Chapter 13. Asynchronous JavaScript

Some computer programs, such as scientific simulations and machine learning models, are compute-bound: they run continuously, without pause, until they have computed their result. Most real-world computer programs, however, are significantly *asynchronous*. This means that they often have to stop computing while waiting for data to arrive or for some event to occur. JavaScript programs in a web browser are typically *event-driven*, meaning that they wait for the user to click or tap before they actually do anything. And JavaScript-based servers typically wait for client requests to arrive over the network before they do anything.

This kind of asynchronous programming is commonplace in JavaScript, and this chapter documents three important language features that help make it easier to work with asynchronous code. Promises, new in ES6, are objects that represent the not-yet-available result of an asynchronous operation. The keywords `async` and `await` were introduced in ES2017 and provide new syntax that simplifies asynchronous programming by allowing you to structure your Promise-based code as if it was synchronous. Finally, asynchronous iterators and the `for/await` loop were introduced in ES2018 and allow you to work with streams of asynchronous events using simple loops that appear synchronous.

Ironically, even though JavaScript provides these powerful features for working with asynchronous code, there are no features of the core language that are themselves asynchronous. In order to demonstrate Promises, `async`, `await`, and `for/await`, therefore, we will first take a detour into client-side and server-side JavaScript to explain some of the asynchronous features of web browsers and Node. (You can learn more about client-side and server-side JavaScript in Chapters 15 and 16.)

13.1 Asynchronous Programming with Callbacks

At its most fundamental level, asynchronous programming in JavaScript is done with *callbacks*. A callback is a function that you write and then pass to some other function. That other function then invokes ("calls back") your function when some condition is met or some (asynchronous)

event occurs. The invocation of the callback function you provide notifies you of the condition or event, and sometimes, the invocation will include function arguments that provide additional details. This is easier to understand with some concrete examples, and the subsections that follow demonstrate various forms of callback-based asynchronous programming using both client-side JavaScript and Node.

13.1.1 Timers

One of the simplest kinds of asynchrony is when you want to run some code after a certain amount of time has elapsed. As we saw in §11.10, you can do this with the `setTimeout()` function:

```
setTimeout(checkForUpdates, 60000);
```

The first argument to `setTimeout()` is a function and the second is a time interval measured in milliseconds. In the preceding code, a hypothetical `checkForUpdates()` function will be called 60,000 milliseconds (1 minute) after the `setTimeout()` call. `checkForUpdates()` is a callback function that your program might define, and `setTimeout()` is the function that you invoke to register your callback function and specify under what asynchronous conditions it should be invoked.

`setTimeout()` calls the specified callback function one time, passing no arguments, and then forgets about it. If you are writing a function that really does check for updates, you probably want it to run repeatedly. You can do this by using `setInterval()` instead of `setTimeout()`:

```
// Call checkForUpdates in one minute and then again every minute after that
let updateIntervalId = setInterval(checkForUpdates, 60000);

// setInterval() returns a value that we can use to stop the repeated
// invocations by calling clearInterval(). (Similarly, setTimeout()
// returns a value that you can pass to clearTimeout())
function stopCheckingForUpdates() {
    clearInterval(updateIntervalId);
}
```

13.1.2 Events

Client-side JavaScript programs are almost universally event driven: rather than running some kind of predetermined computation, they typically wait for the user to do something and then respond to the user's actions. The web browser generates an *event* when the user presses a key on the keyboard, moves the mouse, clicks a mouse button, or touches a touchscreen device. Event-driven JavaScript programs register callback functions for specified types of events in specified contexts, and the web browser invokes those functions whenever the specified events occur. These callback functions are called *event handlers* or *event listeners*, and they are registered with `addEventListener()`:

```
// Ask the web browser to return an object representing the HTML
// <button> element that matches this CSS selector
let okay = document.querySelector('#confirmUpdateDialog button.okay');

// Now register a callback function to be invoked when the user
// clicks on that button.
okay.addEventListener('click', applyUpdate);
```

In this example, `applyUpdate()` is a hypothetical callback function that we assume is implemented somewhere else. The call to `document.querySelector()` returns an object that represents a single specified element in the web page. We call `addEventListener()` on that element to register our callback. Then the first argument to `addEventListener()` is a string that specifies the kind of event we're interested in—a mouse click or touchscreen tap, in this case. If the user clicks or taps on that specific element of the web page, then the browser will invoke our `applyUpdate()` callback function, passing an object that includes details (such as the time and the mouse pointer coordinates) about the event.

13.1.3 Network Events

Another common source of asynchrony in JavaScript programming is network requests. JavaScript running in the browser can fetch data from a web server with code like this:

```
function getCurrentVersionNumber(versionCallback) { // Note callback argumen
    // Make a scripted HTTP request to a backend version API
    let request = new XMLHttpRequest();
    request.open("GET", "http://www.example.com/api/version");
    request.send();
```

```
        // Register a callback that will be invoked when the response arrives
        request.onload = function() {
            if (request.status === 200) {
                // If HTTP status is good, get version number and call callback.
                let currentVersion = parseFloat(request.responseText);
                versionCallback(null, currentVersion);
            } else {
                // Otherwise report an error to the callback
                versionCallback(response.statusText, null);
            }
        };
        // Register another callback that will be invoked for network errors
        request.onerror = request.ontimeout = function(e) {
            versionCallback(e.type, null);
        };
    }
```

Client-side JavaScript code can use the XMLHttpRequest class plus callback functions to make HTTP requests and asynchronously handle the server's response when it arrives.[1] The `getCurrentVersionNumber()` function defined here (we can imagine that it is used by the hypothetical `checkForUpdates()` function we discussed in §13.1.1) makes an HTTP request and defines event handlers that will be invoked when the server's response is received or when a timeout or other error causes the request to fail.

Notice that the code example above does not call `addEventListener()` as our previous example did. For most web APIs (including this one), event handlers can be defined by invoking `addEventListener()` on the object generating the event and passing the name of the event of interest along with the callback function. Typically, though, you can also register a single event listener by assigning it directly to a property of the object. That is what we do in this example code, assigning functions to the `onload`, `onerror`, and `ontimeout` properties. By convention, event listener properties like these always have names that begin with *on*. `addEventListener()` is the more flexible technique because it allows for multiple event handlers. But in cases where you are sure that no other code will need to register a listener for the same object and event type, it can be simpler to simply set the appropriate property to your callback.

Another thing to note about the `getCurrentVersionNumber()` function in this example code is that, because it makes an asynchronous request, it

cannot synchronously return the value (the current version number) that the caller is interested in. Instead, the caller passes a callback function, which is invoked when the result is ready or when an error occurs. In this case, the caller supplies a callback function that expects two arguments. If the XMLHttpRequest works correctly, then `getCurrentVersionNumber()` invokes the callback with a `null` first argument and the version number as the second argument. Or, if an error occurs, then `getCurrentVersionNumber()` invokes the callback with error details in the first argument and `null` as the second argument.

13.1.4 Callbacks and Events in Node

The Node.js server-side JavaScript environment is deeply asynchronous and defines many APIs that use callbacks and events. The default API for reading the contents of a file, for example, is asynchronous and invokes a callback function when the contents of the file have been read:

```
const fs = require("fs"); // The "fs" module has filesystem-related APIs
let options = {           // An object to hold options for our program
    // default options would go here
};

// Read a configuration file, then call the callback function
fs.readFile("config.json", "utf-8", (err, text) => {
    if (err) {
        // If there was an error, display a warning, but continue
        console.warn("Could not read config file:", err);
    } else {
        // Otherwise, parse the file contents and assign to the options obje
        Object.assign(options, JSON.parse(text));
    }

    // In either case, we can now start running the program
    startProgram(options);
});
```

Node's `fs.readFile()` function takes a two-parameter callback as its last argument. It reads the specified file asynchronously and then invokes the callback. If the file was read successfully, it passes the file contents as the second callback argument. If there was an error, it passes the error as the first callback argument. In this example, we express the callback as

an arrow function, which is a succinct and natural syntax for this kind of simple operation.

Node also defines a number of event-based APIs. The following function shows how to make an HTTP request for the contents of a URL in Node. It has two layers of asynchronous code handled with event listeners. Notice that Node uses an `on()` method to register event listeners instead of `addEventListener()`:

```
const https = require("https");

// Read the text content of the URL and asynchronously pass it to the callba
function getText(url, callback) {
    // Start an HTTP GET request for the URL
    request = https.get(url);

    // Register a function to handle the "response" event.
    request.on("response", response => {
        // The response event means that response headers have been received
        let httpStatus = response.statusCode;

        // The body of the HTTP response has not been received yet.
        // So we register more event handlers to to be called when it arrive
        response.setEncoding("utf-8");  // We're expecting Unicode text
        let body = "";                  // which we will accumulate here.

        // This event handler is called when a chunk of the body is ready
        response.on("data", chunk => { body += chunk; });

        // This event handler is called when the response is complete
        response.on("end", () => {
            if (httpStatus === 200) {   // If the HTTP response was good
                callback(null, body);   // Pass response body to the callbac
            } else {                    // Otherwise pass an error
                callback(httpStatus, null);
            }
        });
    });

    // We also register an event handler for lower-level network errors
    request.on("error", (err) => {
        callback(err, null);
    });
}
```

13.2 Promises

Now that we've seen examples of callback and event-based asynchronous programming in client-side and server-side JavaScript environments, we can introduce *Promises*, a core language feature designed to simplify asynchronous programming.

A Promise is an object that represents the result of an asynchronous computation. That result may or may not be ready yet, and the Promise API is intentionally vague about this: there is no way to synchronously get the value of a Promise; you can only ask the Promise to call a callback function when the value is ready. If you are defining an asynchronous API like the `getText()` function in the previous section, but want to make it Promise-based, omit the callback argument, and instead return a Promise object. The caller can then register one or more callbacks on this Promise object, and they will be invoked when the asynchronous computation is done.

So, at the simplest level, Promises are just a different way of working with callbacks. However, there are practical benefits to using them. One real problem with callback-based asynchronous programming is that it is common to end up with callbacks inside callbacks inside callbacks, with lines of code so highly indented that it is difficult to read. Promises allow this kind of nested callback to be re-expressed as a more linear *Promise chain* that tends to be easier to read and easier to reason about.

Another problem with callbacks is that they can make handling errors difficult. If an asynchronous function (or an asynchronously invoked callback) throws an exception, there is no way for that exception to propagate back to the initiator of the asynchronous operation. This is a fundamental fact about asynchronous programming: it breaks exception handling. The alternative is to meticulously track and propagate errors with callback arguments and return values, but this is tedious and difficult to get right. Promises help here by standardizing a way to handle errors and providing a way for errors to propagate correctly through a chain of promises.

Note that Promises represent the future results of single asynchronous computations. They cannot be used to represent repeated asynchronous computations, however. Later in this chapter, we'll write a Promise-based alternative to the `setTimeout()` function, for example. But we can't use

Promises to replace `setInterval()` because that function invokes a callback function repeatedly, which is something that Promises are just not designed to do. Similarly, we could use a Promise instead of the "load" event handler of an XMLHttpRequest object, since that callback is only ever called once. But we typically would not use a Promise in place of a "click" event handler of an HTML button object, since we normally want to allow the user to click a button multiple times.

The subsections that follow will:

- Explain Promise terminology and show basic Promise usage
- Show how promises can be chained
- Demonstrate how to create your own Promise-based APIs

> **IMPORTANT**
>
> Promises seem simple at first, and the basic use case for Promises is, in fact, straightforward and simple. But they can become surprisingly confusing for anything beyond the simplest use cases. Promises are a powerful idiom for asynchronous programming, but you need to understand them deeply to use them correctly and confidently. It is worth taking the time to develop that deep understanding, however, and I urge you to study this long chapter carefully.

13.2.1 Using Promises

With the advent of Promises in the core JavaScript language, web browsers have begun to implement Promise-based APIs. In the previous section, we implemented a `getText()` function that made an asynchronous HTTP request and passed the body of the HTTP response to a specified callback function as a string. Imagine a variant of this function, `getJSON()`, which parses the body of the HTTP response as JSON and returns a Promise instead of accepting a callback argument. We will implement a `getJSON()` function later in this chapter, but for now, let's look at how we would use this Promise-returning utility function:

```
getJSON(url).then(jsonData => {
    // This is a callback function that will be asynchronously
    // invoked with the parsed JSON value when it becomes available.
});
```

`getJSON()` starts an asynchronous HTTP request for the URL you specify and then, while that request is pending, it returns a Promise object. The Promise object defines a `then()` instance method. Instead of passing our callback function directly to `getJSON()`, we instead pass it to the `then()` method. When the HTTP response arrives, the body of that response is parsed as JSON, and the resulting parsed value is passed to the function that we passed to `then()`.

You can think of the `then()` method as a callback registration method like the `addEventListener()` method used for registering event handlers in client-side JavaScript. If you call the `then()` method of a Promise object multiple times, each of the functions you specify will be called when the promised computation is complete.

Unlike many event listeners, though, a Promise represents a single computation, and each function registered with `then()` will be invoked only once. It is worth noting that the function you pass to `then()` is invoked asynchronously, even if the asynchronous computation is already complete when you call `then()`.

At a simple syntactical level, the `then()` method is the distinctive feature of Promises, and it is idiomatic to append `.then()` directly to the function invocation that returns the Promise, without the intermediate step of assigning the Promise object to a variable.

It is also idiomatic to name functions that return Promises and functions that use the results of Promises with verbs, and these idioms lead to code that is particularly easy to read:

```
// Suppose you have a function like this to display a user profile
function displayUserProfile(profile) { /* implementation omitted */ }

// Here's how you might use that function with a Promise.
// Notice how this line of code reads almost like an English sentence:
getJSON("/api/user/profile").then(displayUserProfile);
```

Handling errors with Promises

Asynchronous operations, particularly those that involve networking, can typically fail in a number of ways, and robust code has to be written to handle the errors that will inevitably occur.

For Promises, we can do this by passing a second function to the `then()` method:

```
getJSON("/api/user/profile").then(displayUserProfile, handleProfileError);
```

A Promise represents the future result of an asynchronous computation that occurs after the Promise object is created. Because the computation is performed after the Promise object is returned to us, there is no way that the computation can traditionally return a value or throw an exception that we can catch. The functions that we pass to `then()` provide alternatives. When a synchronous computation completes normally, it simply returns its result to its caller. When a Promise-based asynchronous computation completes normally, it passes its result to the function that is the first argument to `then()`.

When something goes wrong in a synchronous computation, it throws an exception that propagates up the call stack until there is a `catch` clause to handle it. When an asynchronous computation runs, its caller is no longer on the stack, so if something goes wrong, it is simply not possible to throw an exception back to the caller.

Instead, Promise-based asynchronous computations pass the exception (typically as an Error object of some kind, though this is not required) to the second function passed to `then()`. So, in the code above, if `getJSON()` runs normally, it passes its result to `displayUserProfile()`. If there is an error (the user is not logged in, the server is down, the user's internet connection dropped, the request timed out, etc.), then `getJSON()` passes an Error object to `handleProfileError()`.

In practice, it is rare to see two functions passed to `then()`. There is a better and more idiomatic way of handling errors when working with Promises. To understand it, first consider what happens if `getJSON()` completes normally but an error occurs in `displayUserProfile()`. That callback function is invoked asynchronously when `getJSON()` returns, so it is also asynchronous and cannot meaningfully throw an exception (because there is no code on the call stack to handle it).

The more idiomatic way to handle errors in this code looks like this:

```
getJSON("/api/user/profile").then(displayUserProfile).catch(handleProfileErr
```

With this code, a normal result from `getJSON()` is still passed to `displayUserProfile()`, but any error in `getJSON()` or in `displayUserProfile()` (including any exceptions thrown by `displayUserProfile`) get passed to `handleProfileError()`. The `catch()` method is just a shorthand for calling `then()` with a `null` first argument and the specified error handler function as the second argument.

We'll have more to say about `catch()` and this error-handling idiom when we discuss Promise chains in the next section.

PROMISE TERMINOLOGY

Before we discuss Promises further, it is worth pausing to define some terms. When we are not programming and we talk about human promises, we say that a promise is "kept" or "broken." When discussing JavaScript Promises, the equivalent terms are "fulfilled" and "rejected." Imagine that you have called the `then()` method of a Promise and have passed two callback functions to it. We say that the promise has been *fulfilled* if and when the first callback is called. And we say that the Promise has been *rejected* if and when the second callback is called. If a Promise is neither fulfilled nor rejected, then it is *pending*. And once a promise is fulfilled or rejected, we say that it is *settled*. Note that a Promise can never be both fulfilled *and* rejected. Once a Promise settles, it will never change from fulfilled to rejected or vice versa.

Remember how we defined Promises at the start of this section: "a Promise is an object that represents the *result* of an asynchronous operation." It is important to remember that Promises are not just abstract ways registering callbacks to run when some async code finishes—they represent the results of that async code. If the async code runs normally (and the Promise is fulfilled), then that result is essentially the return value of the code. And if the async code does not complete normally (and the Promise is rejected), then the result is an Error object or some other value that the code might have thrown if it was not asynchronous. Any Promise that has settled has a value associated with it, and that value will not change. If the Promise is fulfilled, then the value is a return value that gets passed to any callback functions registered as the first argument of `then()`. If the Promise is rejected, then the value is an error of some sort that is passed to any callback functions registered with `catch()` or as the second argument of `then()`.

The reason that I want to be precise about Promise terminology is that Promises can also be *resolved*. It is easy to confuse this resolved state with the fulfilled state or with settled state, but it is not precisely the same as either. Understanding the resolved state is one of the keys to a deep understanding of Promises, and I'll come back to it after we've discussed Promise chains below.

13.2.2 Chaining Promises

One of the most important benefits of Promises is that they provide a natural way to express a sequence of asynchronous operations as a linear chain of `then()` method invocations, without having to nest each operation within the callback of the previous one. Here, for example, is a hypothetical Promise chain:

```
fetch(documentURL)                          // Make an HTTP request
    .then(response => response.json())      // Ask for the JSON body of the resp
    .then(document => {                     // When we get the parsed JSON
        return render(document);            // display the document to the user
    })
    .then(rendered => {                     // When we get the rendered document
        cacheInDatabase(rendered);          // cache it in the local database.
    })
    .catch(error => handle(error));         // Handle any errors that occur
```

This code illustrates how a chain of Promises can make it easy to express a sequence of asynchronous operations. We're not going to discuss this particular Promise chain at all, however. We will continue to explore the idea of using Promise chains to make HTTP requests, however.

Earlier in this chapter, we saw the XMLHttpRequest object used to make an HTTP request in JavaScript. That strangely named object has an old and awkward API, and it has largely been replaced by the newer, Promise-based Fetch API (§15.11.1). In its simplest form, this new HTTP API is just the function `fetch()`. You pass it a URL, and it returns a Promise. That promise is fulfilled when the HTTP response begins to arrive and the HTTP status and headers are available:

```
fetch("/api/user/profile").then(response => {
    // When the promise resolves, we have status and headers
    if (response.ok &&
        response.headers.get("Content-Type") === "application/json") {
        // What can we do here? We don't actually have the response body yet
    }
});
```

When the Promise returned by `fetch()` is fulfilled, it passes a Response object to the function you passed to its `then()` method. This response object gives you access to request status and headers, and it also defines methods like `text()` and `json()`, which give you access to the body of the response in text and JSON-parsed forms, respectively. But although

the initial Promise is fulfilled, the body of the response may not yet have arrived. So these `text()` and `json()` methods for accessing the body of the response themselves return Promises. Here's a naive way of using `fetch()` and the `response.json()` method to get the body of an HTTP response:

```
fetch("/api/user/profile").then(response => {
    response.json().then(profile => {  // Ask for the JSON-parsed body
        // When the body of the response arrives, it will be automatically
        // parsed as JSON and passed to this function.
        displayUserProfile(profile);
    });
});
```

This is a naive way to use Promises because we nested them, like callbacks, which defeats the purpose. The preferred idiom is to use Promises in a sequential chain with code like this:

```
fetch("/api/user/profile")
    .then(response => {
        return response.json();
    })
    .then(profile => {
        displayUserProfile(profile);
    });
```

Let's look at the method invocations in this code, ignoring the arguments that are passed to the methods:

```
fetch().then().then()
```

When more than one method is invoked in a single expression like this, we call it a *method chain*. We know that the `fetch()` function returns a Promise object, and we can see that the first `.then()` in this chain invokes a method on that returned Promise object. But there is a second `.then()` in the chain, which means that the first invocation of the `then()` method must itself return a Promise.

Sometimes, when an API is designed to use this kind of method chaining, there is just a single object, and each method of that object returns the object itself in order to facilitate chaining. That is not how Promises work,

however. When we write a chain of `.then()` invocations, we are not registering multiple callbacks on a single Promise object. Instead, each invocation of the `then()` method returns a new Promise object. That new Promise object is not fulfilled until the function passed to `then()` is complete.

Let's return to a simplified form of the original `fetch()` chain above. If we define the functions passed to the `then()` invocations elsewhere, we might refactor the code to look like this:

```
fetch(theURL)           // task 1; returns promise 1
    .then(callback1)    // task 2; returns promise 2
    .then(callback2);   // task 3; returns promise 3
```

Let's walk through this code in detail:

1. On the first line, `fetch()` is invoked with a URL. It initiates an HTTP GET request for that URL and returns a Promise. We'll call this HTTP request "task 1" and we'll call the Promise "promise 1".
2. On the second line, we invoke the `then()` method of promise 1, passing the `callback1` function that we want to be invoked when promise 1 is fulfilled. The `then()` method stores our callback somewhere, then returns a new Promise. We'll call the new Promise returned at this step "promise 2", and we'll say that "task 2" begins when `callback1` is invoked.
3. On the third line, we invoke the `then()` method of promise 2, passing the `callback2` function we want invoked when promise 2 is fulfilled. This `then()` method remembers our callback and returns yet another Promise. We'll say that "task 3" begins when `callback2` is invoked. We can call this latest Promise "promise 3", but we don't really need a name for it because we won't be using it at all.
4. The previous three steps all happen synchronously when the expression is first executed. Now we have an asynchronous pause while the HTTP request initiated in step 1 is sent out across the internet.
5. Eventually, the HTTP response starts to arrive. The asynchronous part of the `fetch()` call wraps the HTTP status and headers in a Response object and fulfills promise 1 with that Response object as the value.
6. When promise 1 is fulfilled, its value (the Response object) is passed to our `callback1()` function, and task 2 begins. The job of this task, given a Response object as input, is to obtain the response body as a JSON object.

7. Let's assume that task 2 completes normally and is able to parse the body of the HTTP response to produce a JSON object. This JSON object is used to fulfill promise 2.
8. The value that fulfills promise 2 becomes the input to task 3 when it is passed to the `callback2()` function. This third task now displays the data to the user in some unspecified way. When task 3 is complete (assuming it completes normally), then promise 3 will be fulfilled. But because we never did anything with promise 3, nothing happens when that Promise settles, and the chain of asynchronous computation ends at this point.

13.2.3 Resolving Promises

While explaining the URL-fetching Promise chain with the list in the last section, we talked about promises 1, 2, and 3. But there is actually a fourth Promise object involved as well, and this brings us to our important discussion of what it means for a Promise to be "resolved."

Remember that `fetch()` returns a Promise object which, when fulfilled, passes a Response object to the callback function we register. This Response object has `.text()`, `.json()`, and other methods to request the body of the HTTP response in various forms. But since the body may not yet have arrived, these methods must return Promise objects. In the example we've been studying, "task 2" calls the `.json()` method and returns its value. This is the fourth Promise object, and it is the return value of the `callback1()` function.

Let's rewrite the URL-fetching code one more time in a verbose and nonidiomatic way that makes the callbacks and promises explicit:

```
function c1(response) {                 // callback 1
    let p4 = response.json();
    return p4;                          // returns promise 4
}

function c2(profile) {                  // callback 2
    displayUserProfile(profile);
}

let p1 = fetch("/api/user/profile");    // promise 1, task 1
let p2 = p1.then(c1);                   // promise 2, task 2
let p3 = p2.then(c2);                   // promise 3, task 3
```

In order for Promise chains to work usefully, the output of task 2 must become the input to task 3. And in the example we're considering here, the input to task 3 is the body of the URL that was fetched, parsed as a JSON object. But, as we've just discussed, the return value of callback `c1` is not a JSON object, but Promise `p4` for that JSON object. This seems like a contradiction, but it is not: when `p1` is fulfilled, `c1` is invoked, and task 2 begins. And when `p2` is fulfilled, `c2` is invoked, and task 3 begins. But just because task 2 begins when `c1` is invoked, it does not mean that task 2 must end when `c1` returns. Promises are about managing asynchronous tasks, after all, and if task 2 is asynchronous (which it is, in this case), then that task will not be complete by the time the callback returns.

We are now ready to discuss the final detail that you need to understand to really master Promises. When you pass a callback `c` to the `then()` method, `then()` returns a Promise `p` and arranges to asynchronously invoke `c` at some later time. The callback performs some computation and returns a value `v`. When the callback returns, `p` is *resolved* with the value `v`. When a Promise is resolved with a value that is not itself a Promise, it is immediately fulfilled with that value. So if `c` returns a non-Promise, that return value becomes the value of `p`, `p` is fulfilled and we are done. But if the return value `v` is itself a Promise, then `p` is *resolved but not yet fulfilled*. At this stage, `p` cannot settle until the Promise `v` settles. If `v` is fulfilled, then `p` will be fulfilled to the same value. If `v` is rejected, then `p` will be rejected for the same reason. This is what the "resolved" state of a Promise means: the Promise has become associated with, or "locked onto," another Promise. We don't know yet whether `p` will be fulfilled or rejected, but our callback `c` no longer has any control over that. `p` is "resolved" in the sense that its fate now depends entirely on what happens to Promise `v`.

Let's bring this back to our URL-fetching example. When `c1` returns `p4`, `p2` is resolved. But being resolved is not the same as being fulfilled, so task 3 does not begin yet. When the full body of the HTTP response becomes available, then the `.json()` method can parse it and use that parsed value to fulfill `p4`. When `p4` is fulfilled, `p2` is automatically fulfilled as well, with the same parsed JSON value. At this point, the parsed JSON object is passed to `c2`, and task 3 begins.

This can be one of the trickiest parts of JavaScript to understand, and you may need to read this section more than once. Figure 13-1 presents the process in visual form and may help clarify it for you.

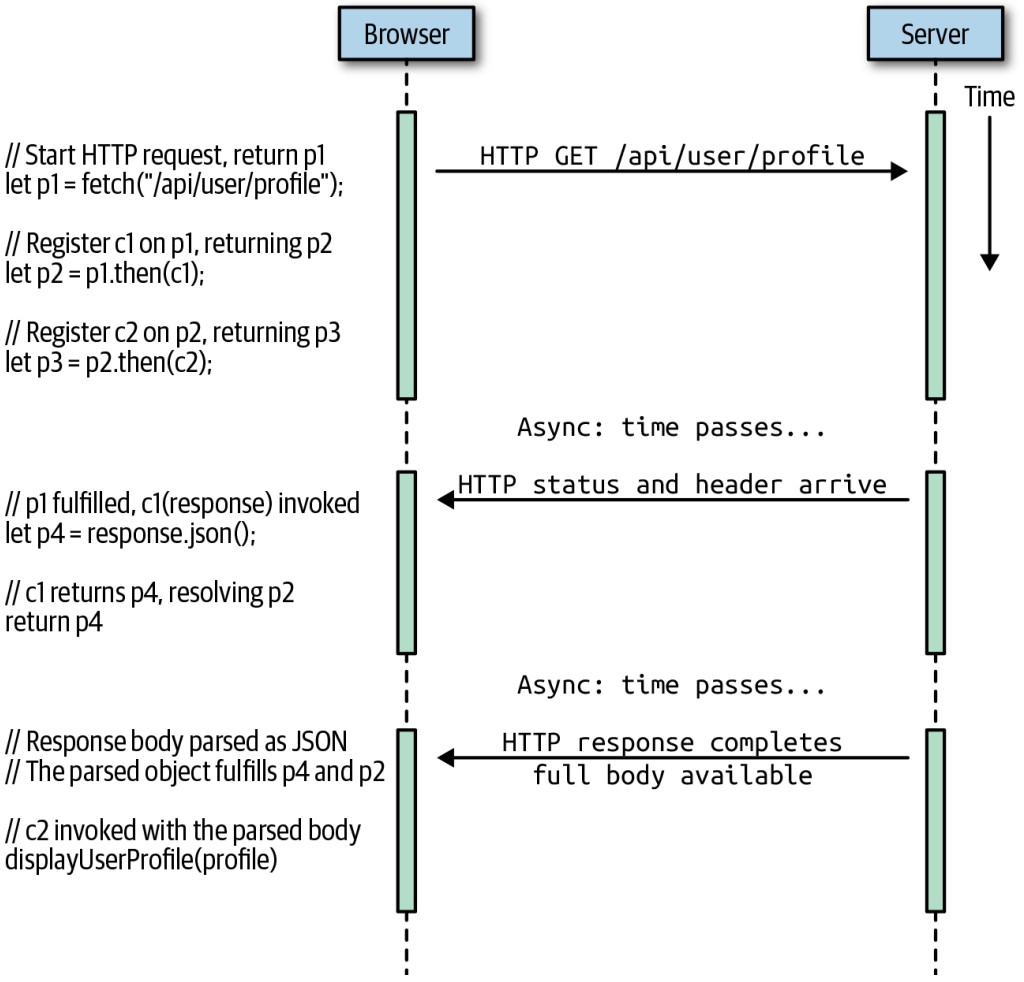

Figure 13-1. Fetching a URL with Promises

13.2.4 More on Promises and Errors

Earlier in the chapter, we saw that you can pass a second callback function to the `.then()` method and that this second function will be invoked if the Promise is rejected. When that happens, the argument to this second callback function is a value—typically an Error object—that represents the reason for the rejection. We also learned that it is uncommon (and even unidiomatic) to pass two callbacks to a `.then()` method. Instead, Promise-related errors are typically handled by adding a `.catch()` method invocation to a Promise chain. Now that we have examined Promise chains, we can return to error handling and discuss it in more detail. To preface the discussion, I'd like to stress that careful error handling is really important when doing asynchronous programming. With synchronous code, if you leave out error-handling code, you'll at least get an exception and a stack trace that you can use to figure out what is going wrong. With asynchronous code, unhandled exceptions will often go unreported, and errors can occur silently, making them much harder to debug. The good news is that the `.catch()` method makes it easy to handle errors when working with Promises.

The catch and finally methods

The `.catch()` method of a Promise is simply a shorthand way to call `.then()` with `null` as the first argument and an error-handling callback as the second argument. Given any Promise `p` and a callback `c`, the following two lines are equivalent:

```
p.then(null, c);
p.catch(c);
```

The `.catch()` shorthand is preferred because it is simpler and because the name matches the `catch` clause in a `try/catch` exception-handling statement. As we've discussed, normal exceptions don't work with asynchronous code. The `.catch()` method of Promises is an alternative that does work for asynchronous code. When something goes wrong in synchronous code, we can speak of an exception "bubbling up the call stack" until it finds a `catch` block. With an asynchronous chain of Promises, the comparable metaphor might be of an error "trickling down the chain" until it finds a `.catch()` invocation.

In ES2018, Promise objects also define a `.finally()` method whose purpose is similar to the `finally` clause in a `try/catch/finally` statement. If you add a `.finally()` invocation to your Promise chain, then the callback you pass to `.finally()` will be invoked when the Promise you called it on settles. Your callback will be invoked if the Promise fulfills or rejects, and it will not be passed any arguments, so you can't find out whether it fulfilled or rejected. But if you need to run some kind of cleanup code (such as closing open files or network connections) in either case, a `.finally()` callback is the ideal way to do that. Like `.then()` and `.catch()`, `.finally()` returns a new Promise object. The return value of a `.finally()` callback is generally ignored, and the Promise returned by `.finally()` will typically resolve or reject with the same value that the Promise that `.finally()` was invoked on resolves or rejects with. If a `.finally()` callback throws an exception, however, then the Promise returned by `.finally()` will reject with that value.

The URL-fetching code that we studied in the previous sections did not do any error handling. Let's correct that now with a more realistic version of the code:

```
fetch("/api/user/profile")     // Start the HTTP request
    .then(response => {        // Call this when status and headers are ready
        if (!response.ok) {    // If we got a 404 Not Found or similar error
            return null;       // Maybe user is logged out; return null profi
        }

        // Now check the headers to ensure that the server sent us JSON.
        // If not, our server is broken, and this is a serious error!
        let type = response.headers.get("content-type");
        if (type !== "application/json") {
            throw new TypeError(`Expected JSON, got ${type}`);
        }

        // If we get here, then we got a 2xx status and a JSON content-type
        // so we can confidently return a Promise for the response
        // body as a JSON object.
        return response.json();
    })
    .then(profile => {         // Called with the parsed response body or nul
        if (profile) {
            displayUserProfile(profile);
        }
        else { // If we got a 404 error above and returned null we end up he
            displayLoggedOutProfilePage();
        }
    })
    .catch(e => {
        if (e instanceof NetworkError) {
            // fetch() can fail this way if the internet connection is down
            displayErrorMessage("Check your internet connection.");
        }
        else if (e instanceof TypeError) {
            // This happens if we throw TypeError above
            displayErrorMessage("Something is wrong with our server!");
        }
        else {
            // This must be some kind of unanticipated error
            console.error(e);
        }
    });
```

Let's analyze this code by looking at what happens when things go wrong. We'll use the naming scheme we used before: `p1` is the Promise returned by the `fetch()` call. `p2` is the Promise returned by the first `.then()` call, and `c1` is the callback that we pass to that `.then()` call. `p3` is the Promise returned by the second `.then()` call, and `c2` is the callback we

pass to that call. Finally, `c3` is the callback that we pass to the `.catch()` call. (That call returns a Promise, but we don't need to refer to it by name.)

The first thing that could fail is the `fetch()` request itself. If the network connection is down (or for some other reason an HTTP request cannot be made), then Promise `p1` will be rejected with a NetworkError object. We didn't pass an error-handling callback function as the second argument to the `.then()` call, so `p2` rejects as well with the same NetworkError object. (If we had passed an error handler to that first `.then()` call, the error handler would be invoked, and if it returned normally, `p2` would be resolved and/or fulfilled with the return value from that handler.) Without a handler, though, `p2` is rejected, and then `p3` is rejected for the same reason. At this point, the `c3` error-handling callback is called, and the NetworkError-specific code within it runs.

Another way our code could fail is if our HTTP request returns a 404 Not Found or another HTTP error. These are valid HTTP responses, so the `fetch()` call does not consider them errors. `fetch()` encapsulates a 404 Not Found in a Response object and fulfills `p1` with that object, causing `c1` to be invoked. Our code in `c1` checks the `ok` property of the Response object to detect that it has not received a normal HTTP response and handles that case by simply returning `null`. Because this return value is not a Promise, it fulfills `p2` right away, and `c2` is invoked with this value. Our code in `c2` explicitly checks for and handles falsy values by displaying a different result to the user. This is a case where we treat an abnormal condition as a nonerror and handle it without actually using an error handler.

A more serious error occurs in `c1` if the we get a normal HTTP response code but the Content-Type header is not set appropriately. Our code expects a JSON-formatted response, so if the server is sending us HTML, XML, or plain text instead, we're going to have a problem. `c1` includes code to check the Content-Type header. If the header is wrong, it treats this as a nonrecoverable problem and throws a TypeError. When a callback passed to `.then()` (or `.catch()`) throws a value, the Promise that was the return value of the `.then()` call is rejected with that thrown value. In this case, the code in `c1` that raises a TypeError causes `p2` to be rejected with that TypeError object. Since we did not specify an error handler for `p2`, `p3` will be rejected as well. `c2` will not be called, and

the TypeError will be passed to `c3`, which has code to explicitly check for and handle this type of error.

There are a couple of things worth noting about this code. First, notice that the error object thrown with a regular, synchronous `throw` statement ends up being handled asynchronously with a `.catch()` method invocation in a Promise chain. This should make it clear why this shorthand method is preferred over passing a second argument to `.then()`, and also why it is so idiomatic to end Promise chains with a `.catch()` call.

Before we leave the topic of error handling, I want to point out that, although it is idiomatic to end every Promise chain with a `.catch()` to clean up (or at least log) any errors that occurred in the chain, it is also perfectly valid to use `.catch()` elsewhere in a Promise chain. If one of the stages in your Promise chain can fail with an error, and if the error is some kind of recoverable error that should not stop the rest of the chain from running, then you can insert a `.catch()` call in the chain, resulting in code that might look like this:

```
startAsyncOperation()
    .then(doStageTwo)
    .catch(recoverFromStageTwoError)
    .then(doStageThree)
    .then(doStageFour)
    .catch(logStageThreeAndFourErrors);
```

Remember that the callback you pass to `.catch()` will only be invoked if the callback at a previous stage throws an error. If the callback returns normally, then the `.catch()` callback will be skipped, and the return value of the previous callback will become the input to the next `.then()` callback. Also remember that `.catch()` callbacks are not just for reporting errors, but for handling and recovering from errors. Once an error has been passed to a `.catch()` callback, it stops propagating down the Promise chain. A `.catch()` callback can throw a new error, but if it returns normally, than that return value is used to resolve and/or fulfill the associated Promise, and the error stops propagating.

Let's be concrete about this: in the preceding code example, if either `startAsyncOperation()` or `doStageTwo()` throws an error, then the `recoverFromStageTwoError()` function will be invoked. If `recoverFromStageTwoError()` returns normally, then its return value

will be passed to `doStageThree()` and the asynchronous operation continues normally. On the other hand, if `recoverFromStageTwoError()` was unable to recover, it will itself throw an error (or it will rethrow the error that it was passed). In this case, neither `doStageThree()` nor `doStageFour()` will be invoked, and the error thrown by `recoverFromStageTwoError()` would be passed to `logStageThreeAndFourErrors()`.

Sometimes, in complex network environments, errors can occur more or less at random, and it can be appropriate to handle those errors by simply retrying the asynchronous request. Imagine you've written a Promise-based operation to query a database:

```
queryDatabase()
    .then(displayTable)
    .catch(displayDatabaseError);
```

Now suppose that transient network load issues are causing this to fail about 1% of the time. A simple solution might be to retry the query with a `.catch()` call:

```
queryDatabase()
    .catch(e => wait(500).then(queryDatabase))  // On failure, wait and retr
    .then(displayTable)
    .catch(displayDatabaseError);
```

If the hypothetical failures are truly random, then adding this one line of code should reduce your error rate from 1% to .01%.

RETURNING FROM A PROMISE CALLBACK

Let's return one last time to the earlier URL-fetching example, and consider the `c1` callback that we passed to the first `.then()` invocation. Notice that there are three ways that `c1` can terminate. It can return normally with the Promise returned by the `.json()` call. This causes `p2` to be resolved, but whether that Promise is fulfilled or rejected depends on what happens with the newly returned Promise. `c1` can also return normally with the value `null`, which causes `p2` to be fulfilled immediately. Finally, `c1` can terminate by throwing an error, which causes `p2` to be rejected. These are the three possible outcomes for a Promise, and the code in `c1` demonstrates how the callback can cause each outcome.

In a Promise chain, the value returned (or thrown) at one stage of the chain becomes the input to the next stage of the chain, so it is critical to get this right. In practice, forgetting to return a value from a callback function is actually a common source of Promise-related bugs, and this is exacerbated by JavaScript's arrow function shortcut syntax. Consider this line of code that we saw earlier:

```
.catch(e => wait(500).then(queryDatabase))
```

Recall from Chapter 8 that arrow functions allow a lot of shortcuts. Since there is exactly one argument (the error value), we can omit the parentheses. Since the body of the function is a single expression, we can omit the curly braces around the function body, and the value of the expression becomes the return value of the function. Because of these shortcuts, the preceding code is correct. But consider this innocuous-seeming change:

```
.catch(e => { wait(500).then(queryDatabase) })
```

By adding the curly braces, we no longer get the automatic return. This function now returns `undefined` instead of returning a Promise, which means that the next stage in this Promise chain will be invoked with `undefined` as its input rather than the result of the retried query. It is a subtle error that may not be easy to debug.

13.2.5 Promises in Parallel

We've spent a lot of time talking about Promise chains for sequentially running the asynchronous steps of a larger asynchronous operation. Sometimes, though, we want to execute a number of asynchronous operations in parallel. The function `Promise.all()` can do this. `Promise.all()` takes an array of Promise objects as its input and returns a Promise. The returned Promise will be rejected if any of the input Promises are rejected. Otherwise, it will be fulfilled with an array of the fulfillment values of each of the input Promises. So, for example, if you want to fetch the text content of multiple URLs, you could use code like this:

```
// We start with an array of URLs
const urls = [ /* zero or more URLs here */ ];
// And convert it to an array of Promise objects
promises = urls.map(url => fetch(url).then(r => r.text()));
// Now get a Promise to run all those Promises in parallel
Promise.all(promises)
    .then(bodies => { /* do something with the array of strings */ })
    .catch(e => console.error(e));
```

`Promise.all()` is slightly more flexible than described before. The input array can contain both Promise objects and non-Promise values. If an element of the array is not a Promise, it is treated as if it is the value of an already fulfilled Promise and is simply copied unchanged into the output array.

The Promise returned by `Promise.all()` rejects when any of the input Promises is rejected. This happens immediately upon the first rejection and can happen while other input Promises are still pending. In ES2020, `Promise.allSettled()` takes an array of input Promises and returns a Promise, just like `Promise.all()` does. But `Promise.allSettled()` never rejects the returned Promise, and it does not fulfill that Promise until all of the input Promises have settled. The Promise resolves to an array of objects, with one object for each input Promise. Each of these returned objects has a `status` property set to "fulfilled" or "rejected." If the status is "fulfilled", then the object will also have a `value` property that gives the fulfillment value. And if the status is "rejected", then the object will also have a `reason` property that gives the error or rejection value of the corresponding Promise:

```
Promise.allSettled([Promise.resolve(1), Promise.reject(2), 3]).then(results
    results[0]  // => { status: "fulfilled", value: 1 }
```

```
    results[1]  // => { status: "rejected", reason: 2 }
    results[2]  // => { status: "fulfilled", value: 3 }
});
```

Occasionally, you may want to run a number of Promises at once but may only care about the value of the first one to fulfill. In that case, you can use `Promise.race()` instead of `Promise.all()`. It returns a Promise that is fulfilled or rejected when the first of the Promises in the input array is fulfilled or rejected. (Or, if there are any non-Promise values in the input array, it simply returns the first of those.)

13.2.6 Making Promises

We've used the Promise-returning function `fetch()` in many of the previous examples because it is one of the simplest functions built in to web browsers that returns a Promise. Our discussion of Promises has also relied on hypothetical Promise-returning functions `getJSON()` and `wait()`. Functions written to return Promises really are quite useful, and this section shows how you can create your own Promise-based APIs. In particular, we'll show implementations of `getJSON()` and `wait()`.

Promises based on other Promises

It is easy to write a function that returns a Promise if you have some other Promise-returning function to start with. Given a Promise, you can always create (and return) a new one by calling `.then()`. So if we use the existing `fetch()` function as a starting point, we can write `getJSON()` like this:

```
function getJSON(url) {
    return fetch(url).then(response => response.json());
}
```

The code is trivial because the Response object of the `fetch()` API has a predefined `json()` method. The `json()` method returns a Promise, which we return from our callback (the callback is an arrow function with a single-expression body, so the return is implicit), so the Promise returned by `getJSON()` resolves to the Promise returned by `response.json()`. When that Promise fulfills, the Promise returned by `getJSON()` fulfills to the same value. Note that there is no error handling in this `getJSON()` implementation. Instead of checking

`response.ok` and the Content-Type header, we instead just allow the `json()` method to reject the Promise it returned with a SyntaxError if the response body cannot be parsed as JSON.

Let's write another Promise-returning function, this time using `getJSON()` as the source of the initial Promise:

```
function getHighScore() {
    return getJSON("/api/user/profile").then(profile => profile.highScore);
}
```

We're assuming that this function is part of some sort of web-based game and that the URL "/api/user/profile" returns a JSON-formatted data structure that includes a `highScore` property.

Promises based on synchronous values

Sometimes, you may need to implement an existing Promise-based API and return a Promise from a function, even though the computation to be performed does not actually require any asynchronous operations. In that case, the static methods `Promise.resolve()` and `Promise.reject()` will do what you want. `Promise.resolve()` takes a value as its single argument and returns a Promise that will immediately (but asynchronously) be fulfilled to that value. Similarly, `Promise.reject()` takes a single argument and returns a Promise that will be rejected with that value as the reason. (To be clear: the Promises returned by these static methods are not already fulfilled or rejected when they are returned, but they will fulfill or reject immediately after the current synchronous chunk of code has finished running. Typically, this happens within a few milliseconds unless there are many pending asynchronous tasks waiting to run.)

Recall from §13.2.3 that a resolved Promise is not the same thing as a fulfilled Promise. When we call `Promise.resolve()`, we typically pass the fulfillment value to create a Promise object that will very soon fulfill to that value. The method is not named `Promise.fulfill()`, however. If you pass a Promise `p1` to `Promise.resolve()`, it will return a new Promise `p2`, which is immediately resolved, but which will not be fulfilled or rejected until `p1` is fulfilled or rejected.

It is possible, but unusual, to write a Promise-based function where the value is computed synchronously and returned asynchronously with `Promise.resolve()`. It is fairly common, however, to have synchronous special cases within an asynchronous function, and you can handle these special cases with `Promise.resolve()` and `Promise.reject()`. In particular, if you detect error conditions (such as bad argument values) before beginning an asynchronous operation, you can report that error by returning a Promise created with `Promise.reject()`. (You could also just throw an error synchronously in that case, but that is considered poor form because then the caller of your function needs to write both a synchronous `catch` clause and use an asynchronous `.catch()` method to handle errors.) Finally, `Promise.resolve()` is sometimes useful to create the initial Promise in a chain of Promises. We'll see a couple of examples that use it this way.

Promises from scratch

For both `getJSON()` and `getHighScore()`, we started off by calling an existing function to get an initial Promise, and created and returned a new Promise by calling the `.then()` method of that initial Promise. But what about writing a Promise-returning function when you can't use another Promise-returning function as the starting point? In that case, you use the `Promise()` constructor to create a new Promise object that you have complete control over. Here's how it works: you invoke the `Promise()` constructor and pass a function as its only argument. The function you pass should be written to expect two parameters, which, by convention, should be named `resolve` and `reject`. The constructor synchronously calls your function with function arguments for the `resolve` and `reject` parameters. After calling your function, the `Promise()` constructor returns the newly created Promise. That returned Promise is under the control of the function you passed to the constructor. That function should perform some asynchronous operation and then call the `resolve` function to resolve or fulfill the returned Promise or call the `reject` function to reject the returned Promise. Your function does not have to be asynchronous: it can call `resolve` or `reject` synchronously, but the Promise will still be resolved, fulfilled, or rejected asynchronously if you do this.

It can be hard to understand the functions passed to a function passed to a constructor by just reading about it, but hopefully some examples will

make this clear. Here's how to write the Promise-based `wait()` function that we used in various examples earlier in the chapter:

```
function wait(duration) {
    // Create and return a new Promise
    return new Promise((resolve, reject) => { // These control the Promise
        // If the argument is invalid, reject the Promise
        if (duration < 0) {
            reject(new Error("Time travel not yet implemented"));
        }
        // Otherwise, wait asynchronously and then resolve the Promise.
        // setTimeout will invoke resolve() with no arguments, which means
        // that the Promise will fulfill with the undefined value.
        setTimeout(resolve, duration);
    });
}
```

Note that the pair of functions that you use to control the fate of a Promise created with the `Promise()` constructor are named `resolve()` and `reject()`, not `fulfill()` and `reject()`. If you pass a Promise to `resolve()`, the returned Promise will resolve to that new Promise. Often, however, you will pass a non-Promise value, which fulfills the returned Promise with that value.

Example 13-1 is another example of using the `Promise()` constructor. This one implements our `getJSON()` function for use in Node, where the `fetch()` API is not built in. Remember that we started this chapter with a discussion of asynchronous callbacks and events. This example uses both callbacks and event handlers and is a good demonstration, therefore, of how we can implement Promise-based APIs on top of other styles of asynchronous programming.

Example 13-1. An asynchronous getJSON() function

```
const http = require("http");

function getJSON(url) {
    // Create and return a new Promise
    return new Promise((resolve, reject) => {
        // Start an HTTP GET request for the specified URL
        request = http.get(url, response => { // called when response starts
            // Reject the Promise if the HTTP status is wrong
            if (response.statusCode !== 200) {
                reject(new Error(`HTTP status ${response.statusCode}`));
```

```
                response.resume();  // so we don't leak memory
            }
            // And reject if the response headers are wrong
            else if (response.headers["content-type"] !== "application/json"
                reject(new Error("Invalid content-type"));
                response.resume();  // don't leak memory
            }
            else {
                // Otherwise, register events to read the body of the respon
                let body = "";
                response.setEncoding("utf-8");
                response.on("data", chunk => { body += chunk; });
                response.on("end", () => {
                    // When the response body is complete, try to parse it
                    try {
                        let parsed = JSON.parse(body);
                        // If it parsed successfully, fulfill the Promise
                        resolve(parsed);
                    } catch(e) {
                        // If parsing failed, reject the Promise
                        reject(e);
                    }
                });
            }
        });
        // We also reject the Promise if the request fails before we
        // even get a response (such as when the network is down)
        request.on("error", error => {
            reject(error);
        });
    });
}
```

13.2.7 Promises in Sequence

`Promise.all()` makes it easy to run an arbitrary number of Promises in parallel. And Promise chains make it easy to express a sequence of a fixed number of Promises. Running an arbitrary number of Promises in sequence is trickier, however. Suppose, for example, that you have an array of URLs to fetch, but that to avoid overloading your network, you want to fetch them one at a time. If the array is of arbitrary length and unknown content, you can't write out a Promise chain in advance, so you need to build one dynamically, with code like this:

```
function fetchSequentially(urls) {
    // We'll store the URL bodies here as we fetch them
    const bodies = [];

    // Here's a Promise-returning function that fetches one body
    function fetchOne(url) {
        return fetch(url)
            .then(response => response.text())
            .then(body => {
                // We save the body to the array, and we're purposely
                // omitting a return value here (returning undefined)
                bodies.push(body);
            });
    }

    // Start with a Promise that will fulfill right away (with value undefin
    let p = Promise.resolve(undefined);

    // Now loop through the desired URLs, building a Promise chain
    // of arbitrary length, fetching one URL at each stage of the chain
    for(url of urls) {
        p = p.then(() => fetchOne(url));
    }

    // When the last Promise in that chain is fulfilled, then the
    // bodies array is ready. So let's return a Promise for that
    // bodies array. Note that we don't include any error handlers:
    // we want to allow errors to propagate to the caller.
    return p.then(() => bodies);
}
```

With this `fetchSequentially()` function defined, we could fetch the URLs one at a time with code much like the fetch-in-parallel code we used earlier to demonstrate `Promise.all()`:

```
fetchSequentially(urls)
    .then(bodies => { /* do something with the array of strings */ })
    .catch(e => console.error(e));
```

The `fetchSequentially()` function starts by creating a Promise that will fulfill immediately after it returns. It then builds a long, linear Promise chain off of that initial Promise and returns the last Promise in the chain. It is like setting up a row of dominoes and then knocking the first one over.

There is another (possibly more elegant) approach that we can take. Rather than creating the Promises in advance, we can have the callback for each Promise create and return the next Promise. That is, instead of creating and chaining a bunch of Promises, we instead create Promises that resolve to other Promises. Rather than creating a domino-like chain of Promises, we are instead creating a sequence of Promises nested one inside the other like a set of matryoshka dolls. With this approach, our code can return the first (outermost) Promise, knowing that it will eventually fulfill (or reject!) to the same value that the last (innermost) Promise in the sequence does. The `promiseSequence()` function that follows is written to be generic and is not specific to URL fetching. It is here at the end of our discussion of Promises because it is complicated. If you've read this chapter carefully, however, I hope you'll be able to understand how it works. In particular, note that the nested function inside `promiseSequence()` appears to call itself recursively, but because the "recursive" call is through a `then()` method, there is not actually any traditional recursion happening:

```
// This function takes an array of input values and a "promiseMaker" functio
// For any input value x in the array, promiseMaker(x) should return a Promi
// that will fulfill to an output value. This function returns a Promise
// that fulfills to an array of the computed output values.
//
// Rather than creating the Promises all at once and letting them run in
// parallel, however, promiseSequence() only runs one Promise at a time
// and does not call promiseMaker() for a value until the previous Promise
// has fulfilled.
function promiseSequence(inputs, promiseMaker) {
    // Make a private copy of the array that we can modify
    inputs = [...inputs];

    // Here's the function that we'll use as a Promise callback
    // This is the pseudorecursive magic that makes this all work.
    function handleNextInput(outputs) {
        if (inputs.length === 0) {
            // If there are no more inputs left, then return the array
            // of outputs, finally fulfilling this Promise and all the
            // previous resolved-but-not-fulfilled Promises.
            return outputs;
        } else {
            // If there are still input values to process, then we'll
            // return a Promise object, resolving the current Promise
            // with the future value from a new Promise.
            let nextInput = inputs.shift(); // Get the next input value,
```

```
                return promiseMaker(nextInput)  // compute the next output value
                    // Then create a new outputs array with the new output value
                    .then(output => outputs.concat(output))
                    // Then "recurse", passing the new, longer, outputs array
                    .then(handleNextInput);
            }
        }

        // Start with a Promise that fulfills to an empty array and use
        // the function above as its callback.
        return Promise.resolve([]).then(handleNextInput);
    }
```

This `promiseSequence()` function is intentionally generic. We can use it to fetch URLs with code like this:

```
// Given a URL, return a Promise that fulfills to the URL body text
function fetchBody(url) { return fetch(url).then(r => r.text()); }
// Use it to sequentially fetch a bunch of URL bodies
promiseSequence(urls, fetchBody)
    .then(bodies => { /* do something with the array of strings */ })
    .catch(console.error);
```

13.3 async and await

ES2017 introduces two new keywords— `async` and `await` —that represent a paradigm shift in asynchronous JavaScript programming. These new keywords dramatically simplify the use of Promises and allow us to write Promise-based, asynchronous code that looks like synchronous code that blocks while waiting for network responses or other asynchronous events. Although it is still important to understand how Promises work, much of their complexity (and sometimes even their very presence!) vanishes when you use them with `async` and `await`.

As we discussed earlier in the chapter, asynchronous code can't return a value or throw an exception the way that regular synchronous code can. And this is why Promises are designed the way the are. The value of a fulfilled Promise is like the return value of a synchronous function. And the value of a rejected Promise is like a value thrown by a synchronous function. This latter similarity is made explicit by the naming of the `.catch()` method. `async` and `await` take efficient, Promise-based code and hide the Promises so that your asynchronous code can be as

easy to read and as easy to reason about as inefficient, blocking, synchronous code.

13.3.1 await Expressions

The `await` keyword takes a Promise and turns it back into a return value or a thrown exception. Given a Promise object `p`, the expression `await p` waits until `p` settles. If `p` fulfills, then the value of `await p` is the fulfillment value of `p`. On the other hand, if `p` is rejected, then the `await p` expression throws the rejection value of `p`. We don't usually use `await` with a variable that holds a Promise; instead, we use it before the invocation of a function that returns a Promise:

```
let response = await fetch("/api/user/profile");
let profile = await response.json();
```

It is critical to understand right away that the `await` keyword does not cause your program to block and literally do nothing until the specified Promise settles. The code remains asynchronous, and the `await` simply disguises this fact. This means that *any code that uses `await` is itself asynchronous*.

13.3.2 async Functions

Because any code that uses `await` is asynchronous, there is one critical rule: *you can only use the `await` keyword within functions that have been declared with the `async` keyword*. Here's a version of the `getHighScore()` function from earlier in the chapter, rewritten to use `async` and `await`:

```
async function getHighScore() {
    let response = await fetch("/api/user/profile");
    let profile = await response.json();
    return profile.highScore;
}
```

Declaring a function `async` means that the return value of the function will be a Promise even if no Promise-related code appears in the body of the function. If an `async` function appears to return normally, then the Promise object that is the real return value of the function will resolve to that apparent return value. And if an `async` function appears to throw

an exception, then the Promise object that it returns will be rejected with that exception.

The `getHighScore()` function is declared `async`, so it returns a Promise. And because it returns a Promise, we can use the `await` keyword with it:

```
displayHighScore(await getHighScore());
```

But remember, that line of code will only work if it is inside another `async` function! You can nest `await` expressions within `async` functions as deeply as you want. But if you're at the top level[2] or are inside a function that is not `async` for some reason, then you can't use `await` and have to deal with a returned Promise in the regular way:

```
getHighScore().then(displayHighScore).catch(console.error);
```

You can use the `async` keyword with any kind of function. It works with the `function` keyword as a statement or as an expression. It works with arrow functions and with the method shortcut form in classes and object literals. (See Chapter 8 for more about the various ways to write functions.)

13.3.3 Awaiting Multiple Promises

Suppose that we've written our `getJSON()` function using `async`:

```
async function getJSON(url) {
    let response = await fetch(url);
    let body = await response.json();
    return body;
}
```

And now suppose that we want to fetch two JSON values with this function:

```
let value1 = await getJSON(url1);
let value2 = await getJSON(url2);
```

The problem with this code is that it is unnecessarily sequential: the fetch of the second URL will not begin until the first fetch is complete. If the second URL does not depend on the value obtained from the first URL, then we should probably try to fetch the two values at the same time. This is a case where the Promise-based nature of `async` functions shows. In order to await a set of concurrently executing `async` functions, we use `Promise.all()` just as we would if working with Promises directly:

```
let [value1, value2] = await Promise.all([getJSON(url1), getJSON(url2)]);
```

13.3.4 Implementation Details

Finally, in order to understand how `async` functions work, it may help to think about what is going on under the hood.

Suppose you write an `async` function like this:

```
async function f(x) { /* body */ }
```

You can think about this as a Promise-returning function wrapped around the body of your original function:

```
function f(x) {
    return new Promise(function(resolve, reject) {
        try {
            resolve((function(x) { /* body */ })(x));
        }
        catch(e) {
            reject(e);
        }
    });
}
```

It is harder to express the `await` keyword in terms of a syntax transformation like this one. But think of the `await` keyword as a marker that breaks a function body up into separate, synchronous chunks. An ES2017 interpreter can break the function body up into a sequence of separate subfunctions, each of which gets passed to the `then()` method of the `await`-marked Promise that precedes it.

13.4 Asynchronous Iteration

We began this chapter with a discussion of callback- and event-based asynchrony, and when we introduced Promises, we noted that they were useful for single-shot asynchronous computations but were not suitable for use with sources of repetitive asynchronous events, such as `setInterval()`, the "click" event in a web browser, or the "data" event on a Node stream. Because single Promises do not work for sequences of asynchronous events, we also cannot use regular `async` functions and the `await` statements for these things.

ES2018 provides a solution, however. Asynchronous iterators are like the iterators described in Chapter 12, but they are Promise-based and are meant to be used with a new form of the `for/of` loop: `for/await`.

13.4.1 The for/await Loop

Node 12 makes its readable streams asynchronously iterable. This means you can read successive chunks of data from a stream with a `for/await` loop like this one:

```
const fs = require("fs");

async function parseFile(filename) {
    let stream = fs.createReadStream(filename, { encoding: "utf-8"});
    for await (let chunk of stream) {
        parseChunk(chunk); // Assume parseChunk() is defined elsewhere
    }
}
```

Like a regular `await` expression, the `for/await` loop is Promise-based. Roughly speaking, the asynchronous iterator produces a Promise and the `for/await` loop waits for that Promise to fulfill, assigns the fulfillment value to the loop variable, and runs the body of the loop. And then it starts over, getting another Promise from the iterator and waiting for that new Promise to fulfill.

Suppose you have an array of URLs:

```
const urls = [url1, url2, url3];
```

You can call `fetch()` on each URL to get an array of Promises:

```
const promises = urls.map(url => fetch(url));
```

We saw earlier in the chapter that we could now use `Promise.all()` to wait for all the Promises in the array to be fulfilled. But suppose we want the results of the first fetch as soon as they become available and don't want to wait for all the URLs to be fetched. (Of course, the first fetch might take longer than any of the others, so this is not necessarily faster than using `Promise.all()`.) Arrays are iterable, so we can iterate through the array of promises with a regular `for/of` loop:

```
for(const promise of promises) {
    response = await promise;
    handle(response);
}
```

This example code uses a regular `for/of` loop with a regular iterator. But because this iterator returns Promises, we can also use the new `for/await` for slightly simpler code:

```
for await (const response of promises) {
    handle(response);
}
```

In this case, the `for/await` loop just builds the `await` call into the loop and makes our code slightly more compact, but the two examples do exactly the same thing. Importantly, both examples will only work if they are within functions declared `async`; a `for/await` loop is no different than a regular `await` expression in that way.

It is important to realize, however, that we're using `for/await` with a regular iterator in this example. Things are more interesting with fully asynchronous iterators.

13.4.2 Asynchronous Iterators

Let's review some terminology from Chapter 12. An *iterable* object is one that can be used with a `for/of` loop. It defines a method with the symbolic name `Symbol.iterator`. This method returns an *iterator* object.

The iterator object has a `next()` method, which can be called repeatedly to obtain the values of the iterable object. The `next()` method of the iterator object returns *iteration result* objects. The iteration result object has a `value` property and/or a `done` property.

Asynchronous iterators are quite similar to regular iterators, but there are two important differences. First, an asynchronously iterable object implements a method with the symbolic name `Symbol.asyncIterator` instead of `Symbol.iterator`. (As we saw earlier, `for/await` is compatible with regular iterable objects but it prefers asynchronously iterable objects, and tries the `Symbol.asyncIterator` method before it tries the `Symbol.iterator` method.) Second, the `next()` method of an asynchronous iterator returns a Promise that resolves to an iterator result object instead of returning an iterator result object directly.

NOTE

In the previous section, when we used `for/await` on a regular, synchronously iterable array of Promises, we were working with synchronous iterator result objects in which the `value` property was a Promise object but the `done` property was synchronous. True asynchronous iterators return Promises for iteration result objects, and both the `value` and the `done` properties are asynchronous. The difference is a subtle one: with asynchronous iterators, the choice about when iteration ends can be made asynchronously.

13.4.3 Asynchronous Generators

As we saw in Chapter 12, the easiest way to implement an iterator is often to use a generator. The same is true for asynchronous iterators, which we can implement with generator functions that we declare `async`. An async generator has the features of async functions and the features of generators: you can use `await` as you would in a regular async function, and you can use `yield` as you would in a regular generator. But values that you `yield` are automatically wrapped in Promises. Even the syntax for async generators is a combination: `async function` and `function *` combine into `async function *`. Here is an example that shows how you might use an async generator and a `for/await` loop to repetitively run code at fixed intervals using loop syntax instead of a `setInterval()` callback function:

```
// A Promise-based wrapper around setTimeout() that we can use await with.
// Returns a Promise that fulfills in the specified number of milliseconds
function elapsedTime(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
}

// An async generator function that increments a counter and yields it
// a specified (or infinite) number of times at a specified interval.
async function* clock(interval, max=Infinity) {
    for(let count = 1; count <= max; count++) { // regular for loop
        await elapsedTime(interval);            // wait for time to pass
        yield count;                            // yield the counter
    }
}

// A test function that uses the async generator with for/await
async function test() {                       // Async so we can use for/awa
    for await (let tick of clock(300, 100)) { // Loop 100 times every 300ms
        console.log(tick);
    }
}
```

13.4.4 Implementing Asynchronous Iterators

Instead of using async generators to implement asynchronous iterators, it is also possible to implement them directly by defining an object with a `Symbol.asyncIterator()` method that returns an object with a `next()` method that returns a Promise that resolves to an iterator result object. In the following code, we re-implement the `clock()` function from the preceding example so that it is not a generator and instead just returns an asynchronously iterable object. Notice that the `next()` method in this example does not explicitly return a Promise; instead, we just declare `next()` to be async:

```
function clock(interval, max=Infinity) {
    // A Promise-ified version of setTimeout that we can use await with.
    // Note that this takes an absolute time instead of an interval.
    function until(time) {
        return new Promise(resolve => setTimeout(resolve, time - Date.now()))
    }

    // Return an asynchronously iterable object
    return {
        startTime: Date.now(),  // Remember when we started
```

```
            count: 1,                // Remember which iteration we're on
            async next() {           // The next() method makes this an iterator
                if (this.count > max) {     // Are we done?
                    return { done: true };  // Iteration result indicating done
                }
                // Figure out when the next iteration should begin,
                let targetTime = this.startTime + this.count * interval;
                // wait until that time,
                await until(targetTime);
                // and return the count value in an iteration result object.
                return { value: this.count++ };
            },
            // This method means that this iterator object is also an iterable.
            [Symbol.asyncIterator]() { return this; }
        };
    }
```

This iterator-based version of the `clock()` function fixes a flaw in the generator-based version. Note that, in this newer code, we target the absolute time at which each iteration should begin and subtract the current time from that in order to compute the interval that we pass to `setTimeout()`. If we use `clock()` with a `for/await` loop, this version will run loop iterations more precisely at the specified interval because it accounts for the time required to actually run the body of the loop. But this fix isn't just about timing accuracy. The `for/await` loop always waits for the Promise returned by one iteration to be fulfilled before it begins the next iteration. But if you use an asynchronous iterator without a `for/await` loop, there is nothing to prevent you from calling the `next()` method whenever you want. With the generator-based version of `clock()`, if you call the `next()` method three times sequentially, you'll get three Promises that will all fulfill at almost exactly the same time, which is probably not what you want. The iterator-based version we've implemented here does not have that problem.

The benefit of asynchronous iterators is that they allow us to represent streams of asynchronous events or data. The `clock()` function discussed previously was fairly simple to write because the source of the asynchrony was the `setTimeout()` calls we were making ourselves. But when we are trying to work with other asynchronous sources, such as the triggering of event handlers, it becomes substantially harder to implement asynchronous iterators—we typically have a single event handler function that responds to events, but each call to the iterator's `next()` method must return a distinct Promise object, and multiple calls to

`next()` may occur before the first Promise resolves. This means that any asynchronous iterator method must be able to maintain an internal queue of Promises that it resolves in order as asynchronous events are occurring. If we encapsulate this Promise-queueing behavior into an AsyncQueue class, then it becomes much easier to write asynchronous iterators based on AsyncQueue.[3]

The AsyncQueue class that follows has `enqueue()` and `dequeue()` methods as you'd expect for a queue class. The `dequeue()` method returns a Promise rather than an actual value, however, which means that it is OK to call `dequeue()` before `enqueue()` has ever been called. The AsyncQueue class is also an asynchronous iterator, and is intended to be used with a `for/await` loop whose body runs once each time a new value is asynchronously enqueued. (AsyncQueue has a `close()` method. Once called, no more values can be enqueued. When a closed queue is empty, the `for/await` loop will stop looping.)

Note that the implementation of AsyncQueue does not use `async` or `await` and instead works directly with Promises. The code is somewhat complicated, and you can use it to test your understanding of the material we've covered in this long chapter. Even if you don't fully understand the AsyncQueue implementation, do take a look at the shorter example that follows it: it implements a simple but very interesting asynchronous iterator on top of AsyncQueue.

```
/**
 * An asynchronously iterable queue class. Add values with enqueue()
 * and remove them with dequeue(). dequeue() returns a Promise, which
 * means that values can be dequeued before they are enqueued. The
 * class implements [Symbol.asyncIterator] and next() so that it can
 * be used with the for/await loop (which will not terminate until
 * the close() method is called.)
 */
class AsyncQueue {
    constructor() {
        // Values that have been queued but not dequeued yet are stored here
        this.values = [];
        // When Promises are dequeued before their corresponding values are
        // queued, the resolve methods for those Promises are stored here.
        this.resolvers = [];
        // Once closed, no more values can be enqueued, and no more unfulfil
        // Promises returned.
        this.closed = false;
    }
```

```
    enqueue(value) {
        if (this.closed) {
            throw new Error("AsyncQueue closed");
        }
        if (this.resolvers.length > 0) {
            // If this value has already been promised, resolve that Promise
            const resolve = this.resolvers.shift();
            resolve(value);
        }
        else {
            // Otherwise, queue it up
            this.values.push(value);
        }
    }

    dequeue() {
        if (this.values.length > 0) {
            // If there is a queued value, return a resolved Promise for it
            const value = this.values.shift();
            return Promise.resolve(value);
        }
        else if (this.closed) {
            // If no queued values and we're closed, return a resolved
            // Promise for the "end-of-stream" marker
            return Promise.resolve(AsyncQueue.EOS);
        }
        else {
            // Otherwise, return an unresolved Promise,
            // queuing the resolver function for later use
            return new Promise((resolve) => { this.resolvers.push(resolve);
        }
    }

    close() {
        // Once the queue is closed, no more values will be enqueued.
        // So resolve any pending Promises with the end-of-stream marker
        while(this.resolvers.length > 0) {
            this.resolvers.shift()(AsyncQueue.EOS);
        }
        this.closed = true;
    }

    // Define the method that makes this class asynchronously iterable
    [Symbol.asyncIterator]() { return this; }

    // Define the method that makes this an asynchronous iterator. The
    // dequeue() Promise resolves to a value or the EOS sentinel if we're
```

```
        // closed. Here, we need to return a Promise that resolves to an
        // iterator result object.
        next() {
            return this.dequeue().then(value => (value === AsyncQueue.EOS)
                                       ? { value: undefined, done: true }
                                       : { value: value, done: false });
        }
    }

    // A sentinel value returned by dequeue() to mark "end of stream" when close
    AsyncQueue.EOS = Symbol("end-of-stream");
```

Because this AsyncQueue class defines the asynchronous iteration basics, we can create our own, more interesting asynchronous iterators simply by asynchronously queueing values. Here's an example that uses AsyncQueue to produce a stream of web browser events that can be handled with a `for/await` loop:

```
    // Push events of the specified type on the specified document element
    // onto an AsyncQueue object, and return the queue for use as an event strea
    function eventStream(elt, type) {
        const q = new AsyncQueue();                  // Create a queue
        elt.addEventListener(type, e=>q.enqueue(e)); // Enqueue events
        return q;
    }

    async function handleKeys() {
        // Get a stream of keypress events and loop once for each one
        for await (const event of eventStream(document, "keypress")) {
            console.log(event.key);
        }
    }
```

13.5 Summary

In this chapter, you have learned:

- Most real-world JavaScript programming is asynchronous.
- Traditionally, asynchrony has been handled with events and callback functions. This can get complicated, however, because you can end up with multiple levels of callbacks nested inside other callbacks, and because it is difficult to do robust error handling.

- Promises provide a new way of structuring callback functions. If used correctly (and unfortunately, Promises are easy to use incorrectly), they can convert asynchronous code that would have been nested into linear chains of `then()` calls where one asynchronous step of a computation follows another. Also, Promises allow you to centralize your error-handling code into a single `catch()` call at the end of a chain of `then()` calls.
- The `async` and `await` keywords allow us to write asynchronous code that is Promise-based under the hood but that looks like synchronous code. This makes the code easier to understand and reason about. If a function is declared `async`, it will implicitly return a Promise. Inside an `async` function, you can `await` a Promise (or a function that returns a Promise) as if the Promise value was synchronously computed.
- Objects that are asynchronously iterable can be used with a `for/await` loop. You can create asynchronously iterable objects by implementing a `[Symbol.asyncIterator]()` method or by invoking an `async function *` generator function. Asynchronous iterators provide an alternative to "data" events on streams in Node and can be used to represent a stream of user input events in client-side JavaScript.

[1] The XMLHttpRequest class has nothing in particular to do with XML. In modern client-side JavaScript, it has largely been replaced by the `fetch()` API, which is covered in §15.11.1. The code example shown here is the last XMLHttpRequest-based example remaining in this book.

[2] You can typically use `await` at the top level in a browser's developer console. And there is a pending proposal to allow top-level `await` in a future version of JavaScript.

[3] I learned about this approach to asynchronous iteration from the blog of Dr. Axel Rauschmayer, *https://2ality.com*.