COMMUNITY EDITION

UNDERSTANDING JAVASCRIPT PROMISES

NICHOLAS C. ZAKAS

Understanding JavaScript Promises

Nicholas C. Zakas

Understanding JavaScript Promises

Nicholas C. Zakas

Contents

Introduction

One of the most powerful aspects of JavaScript is how easily it handles asynchronous programming. As a language created for the web, JavaScript needed to respond to user interactions such as clicks and key presses from the beginning, and so event handlers such as `onclick` were created. Event handlers allowed developers to specify a function to execute at some later point in time in reaction to an event.

Node.js further popularized asynchronous programming in JavaScript by using callback functions in addition to events. As more and more programs started using asynchronous programming, events and callbacks were no longer sufficient to support everything developers wanted to do. *Promises* are the solution to this problem.

Promises are another option for asynchronous programming, and they work like futures and deferreds do in other languages. A promise specifies some code to be executed later (as with events and callbacks) and also explicitly indicates whether the code succeeded or failed at its job. You can chain promises together based on success or failure in ways that make your code easier to understand and debug.

About This Book

The goal of this book is to explain how JavaScript promises work while giving practical examples of when to use them. All new asynchronous JavaScript APIs will be built with promises going forward, and so promises are a central concept to understanding JavaScript as a whole. My hope is that this book will give you the information you need to successfully use promises in your projects.

Browser, Node.js, Deno, and Bun Compatibility

There are multiple JavaScript runtimes that you may use, such as web browsers, Node.js, Deno, and Bun. This book doesn't attempt to address differences between these JavaScript runtimes unless they are so different as to be confusing. In general, this book focuses on promises as described in ECMA-262 and only talks about differences in JavaScript runtimes when they are substantially different. As such, it's possible that your JavaScript runtime may not conform to the standards-based behavior described in this book.

Who This Book Is for

This book is intended as a guide for those who are already familiar with JavaScript. In particular, this book is aimed at intermediate-to-advanced JavaScript developers who work in web browsers, Node.js, or Deno and who want to learn how promises work.

This book is not for beginners who have never written JavaScript. You will need to have a good, basic understanding of the language to make use of this book.

Overview

This book's chapters covers different aspects of JavaScript promises. All chapters include code examples to help you learn new syntax and concepts.

Chapter 1: Promise Basics introduces the concept of promises, how they work, and different ways to create and use them.

Chapter 2: Chaining Promises discusses the various ways to chain multiple promises together to make composing asynchronous operations easier.

Conventions Used

The following typographical conventions are used in this book:

- *Italics* introduces new terms
- `Constant width` indicates a piece of code or filename

All JavaScript code examples are written as modules (also known as ECMAScript modules or ESM).

Additionally, longer code examples are contained in constant width code blocks such as:

```
function doSomething() {
    // empty
}
```

Within a code block, comments to the right of a `console.log()` statement indicate the output you'll see in the browser or Node.js console when the code is executed. For example:

```
console.log("Hi");      // "Hi"
```

If a line of code in a code block throws an error, this is also indicated to the right of the code:

```
doSomething();          // error!
```

Help and Support

If you have questions as you read this book, please send a message to my mailing list: books@humanwhocodes.com. Be sure to mention the title of this book in your subject line.

Acknowledgments

I'm grateful to my father, Speros Zakas, for copyediting this book and for Rob Friesel's technical editing. You both have made this book much better than it was.

Thanks to everyone who reviewed early versions of this book and provided feedback: Mike Sherov, David Hund, Murat Corlu, and Chris Ferdinandi.

About the Author

Nicholas C. Zakas is an independent software engineer, consultant, and coach. He is the creator of the ESLint open source project and serves on the ESLint Technical Steering Committee. Nicholas works with companies and individuals to improve software engineering processes and helps technical leaders grow and succeed. He has also authored or contributed to over a dozen books related to JavaScript and

web development. You can find Nicholas online at humanwhocodes.com and on Twitter @slicknet.

Disclaimer

While the publisher and the author have used good faith effort to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

1. Promise Basics

While promises are often associated with asynchronous operations, they are simply placeholders for values. The value may already be known or, more commonly, the value may be the result of an asynchronous operation. Instead of subscribing to an event or passing a callback to a function, a function can return a promise, like this:

```
// fetch() promises to complete at some point in the future
const promise = fetch("books.json");
```

The `fetch()` function is a common utility function in JavaScript runtimes that makes network requests. The call to `fetch()` doesn't actually complete a network request immediately; that will happen later. Instead, the function returns a promise object (stored in the `promise` variable in this example, but you can name it whatever you want) representing the asynchronous operation so you can work with it in the future. Exactly when you'll be able to work with that result depends entirely on how the promise's lifecycle plays out.

The Promise Lifecycle

Each promise goes through a short lifecycle starting in the *pending* state, which indicates that promise hasn't completed yet. A pending promise is considered *unsettled*. The promise in the previous example is in the pending state as soon as the `fetch()` function returns it. Once the promise completes, the promise is considered *settled* and enters one of two possible states (see Figure 1-1):

1. *Fulfilled*: The promise has completed successfully.
2. *Rejected*: The promise didn't complete successfully due to either an error or some other cause.

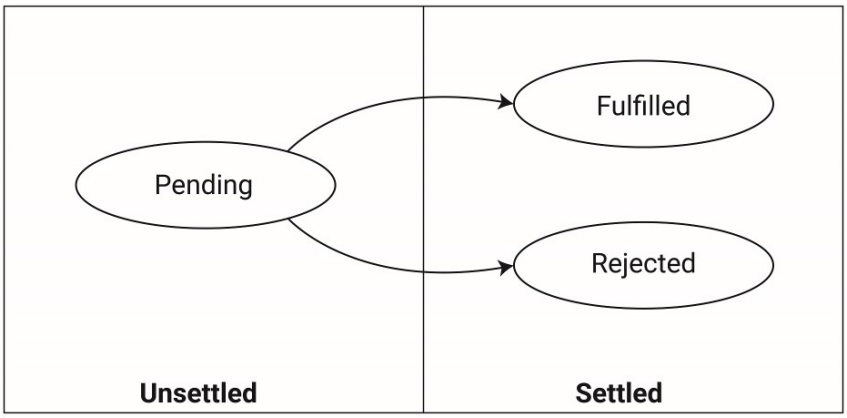

Figure 1-1: Promise states

An internal `[[PromiseState]]` property is set to `"pending"`, `"fulfilled"`, or `"rejected"` to reflect the promise's state. This property isn't exposed on promise objects, so you can't determine which state the promise is in programmatically. But you can take a specific action when a promise changes state by using the `then()` method.

Assigning Handlers with `then()`

The `then()` method is present on all promises and takes two arguments. The first argument is a function to call when the promise is fulfilled, called the *fulfillment handler*. Any additional data related to the asynchronous operation is passed to this function. The second argument is a function to call when the promise is rejected, called the *rejection handler*. Similar to the fulfillment handler, the rejection handler is passed any additional data related to the rejection.

> Any object that implements the `then()` method in this way is called a *thenable*. All promises are thenables, but not all thenables are promises.

Both arguments to `then()` are optional, so you can listen for any combination of fulfillment and rejection. For example, consider this set of `then()` calls:

```
const promise = fetch("books.json");

// add a fulfillment and rejection handler
promise.then(response => {
```

```
    // fulfillment
    console.log(response.status);
}, reason => {
    // rejection
    console.error(reason.message);
});

// add another fulfillment handler
promise.then(response => {
    // fulfillment
    console.log(response.statusText);
});

// add another rejection handler
promise.then(null, reason => {
    // rejection
    console.error(reason.message);
});
```

All three `then()` calls operate on the same promise. The first call assigns both a fulfillment and a rejection handler. The second only assigns a fulfillment handler; errors won't be reported. The third just assigns a rejection handler and doesn't report success.

One quirk of the `fetch()` function is that the returned promise is fulfilled whenever it receives an HTTP status, even 404 or 500. The promise is only rejected when the network request fails for some reason. If you want to ensure that the status is in the 200-299 range, you can check the `response.ok` property, as in this example:

```
const promise = fetch("books.json");

promise.then(response => {
    if (response.ok) {
        console.log("Request succeeded.");
    } else {
        console.error("Request failed.");
    }
});
```

Assigning Rejection Handlers with `catch()`

Promises also have a `catch()` method that behaves the same as `then()` when only a rejection handler is passed. For example, the following `catch()` and `then()` calls are functionally equivalent:

```
const promise = fetch("books.json");

promise.catch(reason => {
    // rejection
    console.error(reason.message);
});

// is the same as:

promise.then(null, reason => {
    // rejection
    console.error(reason.message);
});
```

The intent behind `then()` and `catch()` is for you to use them in combination to clearly indicate how a result is handled. This system is better than events and

callbacks because it makes success or failure completely clear. (Events tend not to fire when there's an error, and in callbacks you must always remember to check the error argument.) Just know that if you don't attach a rejection handler to a promise that is rejected, then the JavaScript runtime will output a message to the console, or throw an error, or both (depending on the runtime).

Assigning Settlement Handlers with `finally()`

To go along with `then()` and `catch()` there is also `finally()`. The callback passed to `finally()` (called a *settlement handler*) is called regardless of success or failure. Unlike the callbacks for `then()` and `catch()`, `finally()` callbacks do not receive any arguments because it isn't clear whether the promise was fulfilled or rejected. Because the settlement handler is called both on fulfillment and rejection, it is similar (but not the same; discussed further in Chapter 2) to passing the handler for both fulfillment and rejection using `then()`. Here's an example:

```
const promise = fetch("books.json");

promise.finally(() => {
    // no way to know if fulfilled or rejected
    console.log("Settled");
});

// is similar to:

const callback = () => {
    console.log("Settled");
};

promise.then(callback, callback);
```

As long as you don't access the argument passed to `callback`, the behavior between these two examples is the same. However, as with `catch()`, using `finally()` makes your intention clearer as compared to `then()`.

Settlement handlers are useful when you want to know that an operation has completed and you don't care about the result. As an example, you may want to display a loading indicator on a web page while a `fetch()` request is active and then hide it when the request is complete. It doesn't matter if the request was successful

or not because the loading indicator should stop once the request is complete. You might have code like this in your web application:

```
const appElement = document.getElementById("app");
const promise = fetch("books.json");

appElement.classList.add("loading");

promise.then(() => {
    // handle success
});

promise.catch(() => {
    // handle failure
});

promise.finally(() => {
    appElement.classList.remove("loading");
});
```

Here, `appElement` represents the HTML element that wraps the entire application on the page. A network request is initiated using `fetch()` and the CSS class `"loading"` is added to the HTML element (allowing you to change any styles as appropriate). When the network request completes, `promise` is settled and the settlement handler removes the `"loading"` class from the HTML element to reset the application state. You can still respond to success and failure using `then()` and `catch()` while `finally()` solely handles the state change. Without `finally()`, you would need to remove the `"loading"` class in both the fulfillment and rejection handlers.

> The settlement handlers added with `finally()` do not prevent rejections from outputting an error to the console or throwing an error. You must still add a rejection handler to prevent the error from being thrown in that case.

Assigning Handlers to Settled Promises

A fulfillment, rejection, or settlement handler will still be executed even if it is added after the promise is already settled. This allows you to add new fulfillment

and rejection handlers at any time and guarantee that they will be called. For example:

```
const promise = fetch("books.json");

// original fulfillment handler
promise.then(response => {
    console.log(response.status);

    // now add another
    promise.then(response => {
        console.log(response.statusText);
    });
});
```

In this code, the fulfillment handler adds another fulfillment handler to the same promise. The promise is already fulfilled at this point, so the new fulfillment handler is added to the microtask queue and called when ready. Rejection and settlement handlers work the same way.

Handlers and Microtasks

JavaScript executed in the regular flow of a program is executed as a *task*, which is to say that the JavaScript runtime has created a new execution context and executes the code completely, exiting when finished. As an example, an `onclick` handler for a button in a web page is executed as a task. When the button is clicked, a new task is created and the `onclick` handler is executed. Once complete, the JavaScript runtime waits for the next interaction to execute more code. Promise handlers, however, are executed in a different way.

All promise handlers, whether fulfillment, rejection, or settlement, are executed as *microtasks* inside of the JavaScript engine. Microtasks are queued and then executed immediately after the currently running task has completed, before the JavaScript runtime becomes idle. Calling `then()`, `catch()`, or `finally()` tells a promise to queue the specified microtasks once the promise is settled.

This is different than creating timers using `setTimeout()` or `setInterval()`, both of which create new tasks to be executed at a later point in time. Queued promise handlers will always execute before timers that are queued in the same task. You

can test this for yourself by using the global `queueMicrotask()` function, which is used to create microtasks outside of promises:

```
setTimeout(() => {
    console.log("timer");

    queueMicrotask(() => {
        console.log("microtask in timer");
    });

}, 0);

queueMicrotask(() => {
    console.log("microtask");
});
```

In this code, a timer is created with a delay of 0 milliseconds, and inside of that timer a new microtask is created. Also, a microtask is created outside of the timer. When this code executes, you will see the following output to the console:

```
microtask
timer
microtask in timer
```

Even though the timer is set for a delay of 0 milliseconds, the microtask executes first, followed by the timer, followed by the microtask inside of the timer.

The most important thing to remember about microtasks, including all promise handlers, is that they are executed as soon as possible once a task is complete. This minimizes the amount of time between a promise settling and the reaction to the settling, making promises suitable for situations where runtime performance is important.

Creating New (Unsettled) Promises

New promises are created using the `Promise` constructor. This constructor accepts a single argument: a function called the *executor*, which contains the code to initialize the promise. The executor is passed two functions named `resolve()` and `reject()` as arguments. You call the `resolve()` function when the executor has

finished successfully to signal that the promise is resolved or the `reject()` function to indicate that the operation has failed.

Here's an example using the old `XMLHttpRequest` browser API:

```
// Browser example

function requestURL(url) {
    return new Promise((resolve, reject) => {

        const xhr = new XMLHttpRequest();

        // assign event handlers
        xhr.addEventListener("load", () => {
            resolve({
                status: xhr.status,
                text: xhr.responseText
            });
        });

        xhr.addEventListener("error", error => {
            reject(error);
        });

        // send the request
        xhr.open("get", url);
        xhr.send();
    });
}

const promise = requestURL("books.json");

// listen for both fulfillment and rejection
promise.then(response => {
    // fulfillment
    console.log(response.status);
    console.log(response.text);
}, reason => {
    // rejection
```

```
    console.error(reason.message);
});
```

In this example, the `XMLHttpRequest` call is wrapped in a promise. The `load` event indicates when a request has completed successfully, and so the promise executor calls `resolve()` in the event handler. Similarly, the `error` event indicates when the request couldn't be completed and so `reject()` is called in that event handler. You can follow this same process (using `resolve()` and `reject()` in event handlers) for converting event-based functionality into promise-based functionality.

One important aspect of executors is that they run immediately upon creation of the promise. In the previous example, the `xhr` object is created, event handlers assigned, and the call initiated before the promise is returned from `requestURL()`. When either `resolve()` or `reject()` is called inside the executor, then the promise's state and value are immediately set, but all promise handlers (being microtasks) will not execute until the current script job completes. For example, consider what happens if you call `resolve()` immediately inside an executor, as in this code:

```
const promise = new Promise((resolve, reject) => {
    console.log("Executor");
    resolve(42);
});

promise.then(result => {
    console.log(result);
});

console.log("Hi!");
```

Here, the promise is resolved immediately without any delay, and then a fulfillment handler is added using `then()` to output the result. Even though the promise is already resolved when the fulfillment handler is added, the output will be as follows:

```
Executor
Hi!
42
```

The executor is run first, outputting `"Executor"` to the console. Next, the fulfillment handler is assigned but is not executed immediately. Instead, a new microtask is created to run after the current script job. That means `console.log("Hi!")` executes before the fulfillment handler, which outputs `42` after the rest of the script has completed.

> A promise can only be resolved once, so if you call `resolve()` more than once inside of an executor, every call after the first is ignored.

Executor Errors

If an error is thrown inside an executor, then the promise's rejection handler is called. For example:

```
const promise = new Promise((resolve, reject) => {
    throw new Error("Uh oh!");
});

promise.catch(reason => {
    console.log(reason.message);    // "Uh oh!"
});
```

In this code, the executor intentionally throws an error. There is an implicit `try-catch` inside every executor so that the error is caught and then passed to the rejection handler. The previous example is equivalent to:

```
const promise = new Promise((resolve, reject) => {
    try {
        throw new Error("Uh oh!");
    } catch (ex) {
        reject(ex);
    }
});

promise.catch(reason => {
    console.log(reason.message);    // "Uh oh!"
});
```

The executor handles catching any thrown errors to simplify this common use case, and just like other rejections, the JavaScript engine throws an error and stops execution if no rejection handler is assigned.

Creating Deferred Promises

The `Promise` constructor is useful for creating new promises when you can easily encapsulate settlement behavior inside of an executor. However, there may be times when you want to create a promise and then define its settlement behavior later. You can create a *deferred promise* using `Promise.withResolvers()` where the settlement behavior occurs after the creation of the promise rather than during its creation.

The `Promise.withResolvers()` method returns an object with the following properties:

- `promise` - an instance of `Promise`
- `resolve` - a function to call to resolve `promise`
- `reject` - a function to call to reject `promise`

This method is most commonly used with a destructuring assignment to extract the individual properties of the returned object, as in this example:

```
const { promise, resolve, reject } = Promise.withResolvers();
```

From there, you can determine when to return `promise` and when to use `resolve` and `reject`.

As an example, here's the `requestURL()` function from earlier in this chapter rewritten to use `Promise.withResolvers()`:

```
function requestURL(url) {

    const { promise, resolve, reject } = Promise.withResolvers();

    const xhr = new XMLHttpRequest();

    // assign event handlers
    xhr.addEventListener("load", () => {
        resolve({
            status: xhr.status,
```

```
            text: xhr.responseText
        });
    });

    xhr.addEventListener("error", error => {
        reject(error);
    });

    // send the request
    xhr.open("get", url);
    xhr.send();

    return promise;
}
```

This version of `requestURL()` is functionally equivalent to the previous one except that it uses a deferred promise. Doing so allows the code that was previously contained in a promise executor to move into the surrounding function.

Many developers prefer using deferred promises to avoid the overhead of executor functions.

Creating Settled Promises

The `Promise` constructor is the best way to create unsettled promises due to the dynamic nature of what the promise executor does. But if you want a promise to represent a previously computed value, then it doesn't make sense to create an executor that simply passes a value to the `resolve()` or `reject()` function. Instead, there are two methods that create settled promises given a specific value.

> Creating settled promises is helpful for compatibility with APIs that expect promises to be passed as arguments.

Using Promise.resolve()

The `Promise.resolve()` method accepts a single argument and returns a promise in the fulfilled state. That means you don't have to supply an executor if you know the value of the promise already. For example:

```
const promise = Promise.resolve(42);

promise.then(value => {
    console.log(value);         // 42
});
```

This code creates a fulfilled promise so the fulfillment handler receives `42` as `value`. As with other examples in this chapter, the fulfillment handler is executed as a microtask after the current script job completes. If a rejection handler were added to this promise, the rejection handler would never be called because the promise will never be in the rejected state.

> If you pass a promise to `Promise.resolve()`, then the function returns the same promise that you passed in. For example:
>
> ```
> const promise1 = Promise.resolve(42);
> const promise2 = Promise.resolve(promise1);
>
> console.log(promise1 === promise2);             // true
> ```

Using Promise.reject()

You can also create rejected promises by using the `Promise.reject()` method. This works like `Promise.resolve()` except the created promise is in the rejected state, as follows:

```
const promise = Promise.reject(42);

promise.catch(reason => {
    console.log(reason);        // 42
});
```

Any additional rejection handlers added to this promise would be called, but fulfillment handlers will not because the promise will never be in the fulfilled state.

> If you pass a promise to either the `Promise.resolve()` or `Promise.reject()` methods, the promise is returned without modification.

Non-Promise Thenables

Both `Promise.resolve()` and `Promise.reject()` also accept non-promise thenables as arguments. When passed a non-promise thenable, these methods create a new promise that is settled with the same value and state of the settled thenable.

A non-promise thenable is created when an object has a `then()` method that accepts a `resolve` and a `reject` argument, like this:

```
const thenable = {
    then(resolve, reject) {
        resolve(42);
    }
};
```

The `thenable` object in this example has no characteristics associated with a promise other than the `then()` method. You can call `Promise.resolve()` to convert `thenable` into a fulfilled promise:

```
const thenable = {
    then(resolve, reject) {
        resolve(42);
    }
};

const promise = Promise.resolve(thenable);
promise.then(value => {
    console.log(value);     // 42
});
```

In this example, `Promise.resolve()` calls `thenable.then()` so that a promise state can be determined. The promise state for `thenable` is fulfilled because `resolve(42)` is called inside the `then()` method. A new promise called `promise` is created in the fulfilled state with the value passed from `thenable` (that is, `42`), and the fulfillment handler for `promise` receives `42` as the value.

The same process can be used with `Promise.resolve()` to create a rejected promise from a thenable:

```
const thenable = {
    then(resolve, reject) {
        reject(42);
    }
};

const promise = Promise.resolve(thenable);
promise.catch(value => {
    console.log(value);     // 42
});
```

This example is similar to the last except that `thenable` is rejected. When `thenable.then()` executes, a new promise is created in the rejected state with a value of `42`. That value is then passed to the rejection handler for `promise`.

`Promise.resolve()` and `Promise.reject()` work like this to allow you to easily work with non-promise thenables. A lot of libraries used thenables prior to promises being introduced in 2015, so the ability to convert thenables into formal promises is important for backwards compatibility with previously existing libraries. When you're unsure if an object is a promise, passing the object through `Promise.resolve()` or `Promise.reject()` (depending on your anticipated result) is the best way to find out because promises just pass through unchanged.

Summary

A promise is a placeholder for a value that may be provided later as the result of some asynchronous operation. Instead of assigning an event handler or passing a callback into a function, you can use a promise to represent the result of an operation.

Promises have three states: pending, fulfilled, and rejected. A promise starts in a pending (unsettled) state and becomes fulfilled on a successful execution or rejected on a failure (fulfillment and rejection are settled states). In either case, handlers can be added to indicate when a promise is settled. The `then()` method allows you to assign a fulfillment and rejection handler; the `catch()` method allows you to assign only a rejection handler; the `finally()` method allows you to assign a settlement handler that is always called regardless of the outcome. All

promise handlers are run as microtasks so they will not execute until the current script job is complete.

One way to create new unsettled promises is using the `Promise` constructor, which accepts an executor function as its only argument. The executor function is passed `resolve()` and `reject()` functions that you use to indicate the success or failure of the promise. The executor runs immediately upon creation of the promise, unlike handlers, which are run as microtasks. Any errors thrown in an executor are automatically caught and passed to `reject()`.

Another way to create new unsettled promises is to call `Promise.withResolvers()`, which returns an object containing three properties: `promise`, `resolve`, and `reject`. Promises creates with `Promise.withResolvers()` are called deferred promises because their settlement behavior is determined after promise creationg, as opposed to using the `Promise` constructor, where the settlement behavior is defined inside the executor function.

It's possible to create settled promises using `Promise.resolve()` for fulfilled promises and `Promise.reject()` for rejected promises. Each method will wrap its argument in a promise (if it's not a promise and not a non-promise thenable), create a new promise (for non-promise thenables), or pass through any existing promise. These methods are helpful when you are unsure if the value is a promise but want it to behave like one.

While creating single promises is a useful and effective way to work with asynchronous operations in JavaScript, promises allow interesting composition patterns when chained together. In the next chapter, you'll learn how promise handlers work to create promise chains and why that's a valuable capability.

2. Chaining Promises

To this point, promises may seem like little more than an incremental improvement over using some combination of a callback and the `setTimeout()` function, but there is much more to promises than meets the eye. More specifically, there are a number of ways to chain promises together to accomplish more complex asynchronous behavior.

Each call to `then()`, `catch()`, or `finally()` actually creates and returns another promise. This second promise is settled only once the first has been fulfilled or rejected. Consider this example:

```
const promise = Promise.resolve(42);

promise.then(value => {
    console.log(value);
}).then(() => {
    console.log("Finished");
});
```

The code outputs:

```
42
Finished
```

The call to `promise.then()` returns a second promise on which `then()` is called. The second `then()` fulfillment handler is only called after the first promise has been resolved. If you unchain this example, it looks like this:

```
const promise1 = Promise.resolve(42);
```

```
const promise2 = promise1.then(value => {
    console.log(value);
});

promise2.then(() => {
    console.log("Finished");
});
```

In this unchained version of the code, the result of `promise1.then()` is stored in `promise2`, and then `promise2.then()` is called to add the final fulfillment handler. The call to `promise2.then()` also returns a promise. This example just doesn't use that promise.

Catching Errors

Promise chaining allows you to catch errors that may occur in a fulfillment or rejection handler from a previous promise. For example:

```
const promise = Promise.resolve(42);

promise.then(value => {
    throw new Error("Oops!");
}).catch(reason => {
    console.error(reason.message);    // "Oops!"
});
```

In this code, the fulfillment handler for `promise` throws an error. The chained call to the `catch()` method, which is on a second promise, is able to receive that error through its rejection handler. The same is true if a rejection handler throws an error:

```
const promise = new Promise((resolve, reject) => {
    throw new Error("Uh oh!");
});

promise.catch(reason => {
    console.log(reason.message);    // "Uh oh!"
    throw new Error("Oops!");
}).catch(reason => {
```

```
    console.error(reason.message);  // "Oops!"
});
```

Here, the executor throws an error that triggers the `promise`'s rejection handler. That handler then throws another error that is caught by the second promise's rejection handler. The chained promise calls are aware of errors in other promises in the chain.

You can use this ability to catch errors through a promise chain to effectively act like a `try-catch` statement. Consider using `fetch()` to retrieve some data and wanting to catch any errors that occur:

```
const promise = fetch("books.json");

promise.then(response => {
    console.log(response.status);
}).catch(reason => {
    console.error(reason.message);
});
```

This example will output the response status from the `fetch()` call if it succeeds and will output the error message if the call fails. You can take this a step further and handle status codes outside of the 200-299 range as errors by checking the `response.ok` property (discussed in Chapter 1) and throwing an error if it is `false`, as in this example:

```
const promise = fetch("books.json");

promise.then(response => {
    if (response.ok) {
        console.log(response.status);
    } else {
        throw new Error(`Unexpected status code: ${
            response.status
        } ${response.statusText}`);
    }
}).catch(reason => {
    console.error(reason.message);
});
```

The chained `catch()` call in this example creates a rejection handler that catches both errors returned by `fetch()` and also any errors thrown in the fulfillment handler. So instead of needing two different handles for catching the two different types of errors, you can use one to handle all of the errors that may occur in the chain.

> Always have a rejection handler at the end of a promise chain to ensure that you can properly handle any errors that may occur.

Using finally() in Promise Chains

The `finally()` method behaves differently than either `then()` or `catch()` in that it copies the state and value of the previous promise into its returned promise. That means if the original promise is fulfilled with a value, then `finally()` returns a promise that is fulfilled with the same value. For example:

```
const promise = Promise.resolve(42);

promise.finally(() => {
    console.log("Finally called.");
}).then(value => {
    console.log(value);         // 42
});
```

Here, the settlement handler can't receive the fulfilled value from `promise`, so that value is copied to a new promise that is returned from the method call. The new promise is fulfilled with the value `42` (copied from `promise`) so the fulfillment handler receives `42` as an argument. Keep in mind that even though the returned promise and `promise` have the same value, they are not the same object, as you can see in this example:

```
const promise1 = Promise.resolve(42);

const promise2 = promise1.finally(() => {
    console.log("Finally called.");
});

promise2.then(value => {
```

```
    console.log(value);         // 42
});

console.log(promise1 === promise2); // false
```

In this code, the returned value from `promise1.finally()` is stored in `promise2`, at which point you can determine that it is not the same object as `promise1`. The call to `finally()` always copies the state and value from the original promise. That also means that when `finally()` is called on a rejected promise, it in turn returns a rejected promise, as in this example:

```
const promise = Promise.reject(43);

promise.finally(() => {
    console.log("Finally called.");
}).catch(reason => {
    console.error(reason);      // 43
});
```

The promise `promise` in this example is rejected with a reason of `43`. Once again, the settlement handler cannot access this information as it is not passed in as an argument, so instead it returns a new promise that is rejected for the same reason. You can then use `catch()` to retrieve the reason.

The one exception to how `finally()` works is when an error is thrown inside of the settlement handler or a rejected promise is returned. In this one case, the returned promise from `finally()` does not maintain the state and value from the original promise, and instead is rejected with the thrown error as the reason. Here's an example:

```
const promise1 = Promise.reject(43);

promise1.finally(() => {
    throw 44;
}).catch(reason => {
    console.error(reason);      // 44
});

const promise2 = Promise.reject(43);
```

```
promise2.finally(() => {
    return Promise.reject(44);
}).catch(reason => {
    console.error(reason);      // 44
});
```

Because the settlement handlers throw `44` or return `Promise.reject(44)` in this example, the returned promise is rejected with the value of `44` and that is output to the console instead of `43`. The state and value of the original promise are lost as a consequence of the error being thrown in the settlement handler.

In Chapter 1, you saw how a settlement handler can be used to toggle the loading state of an application based on a call to `fetch()`. Rewriting that example using promise chains, and mixing in some error handling from earlier in this chapter, here's a complete example:

```
const appElement = document.getElementById("app");
const promise = fetch("books.json");

appElement.classList.add("loading");

promise.then(response => {
    if (response.ok) {
        console.log(response.status);
    } else {
        throw new Error(`Unexpected status code: ${
            response.status
        } ${response.statusText}`);
    }
}).finally(() => {
    appElement.classList.remove("loading");
}).catch(reason => {
    console.error(reason.message);
});
```

Unlike a `try-catch` statement, you don't want `finally()` to be the last part of the chain just in case it throws an error. So `then()` is called first, to handle the response from `fetch()`, then `finally()` is added to the chain to trigger the UI change, and

last `catch()` adds the error handler for the entire chain. This is where settlement handlers passing along the state of the previous promise is helpful: if the fulfillment handler ends up throwing an error, the settlement handler will pass that rejection state along so the rejection handler can access it.

Returning Values in Promise Chains

Another important aspect of promise chains is the ability to pass data from one promise to the next. You've already seen that a value passed to the `resolve()` handler inside an executor is passed to the fulfillment handler for that promise. You can continue passing data along a chain by specifying a return value from the fulfillment handler. For example:

```
const promise = Promise.resolve(42);

promise.then(value => {
    console.log(value);         // 42
    return value + 1;
}).then(value => {
    console.log(value);         // 43
});
```

The fulfillment handler for `promise` returns `value + 1` when executed. Since `value` is `42` (from the executor), the fulfillment handler returns `43`. That value is then passed to the fulfillment handler of the second promise, which outputs it to the console.

You could do the same thing with the rejection handler. When a rejection handler is called, it may return a value. If it does, that value is used to fulfill the next promise in the chain, like this:

```
const promise = Promise.reject(42);

promise.catch(value => {
    // rejection handler
    console.error(value);       // 42
    return value + 1;
}).then(value => {
    // fulfillment handler
```

```
    console.log(value);         // 43
});
```

Here, a rejected promise is created with a value of `42`. That value is passed into the rejection handler for the promise, where `value + 1` is returned. Even though this return value is coming from a rejection handler, it is still used in the fulfillment handler of the next promise in the chain. The failure of one promise can allow recovery of the entire chain if necessary.

Using `finally()`, however, results in a different behavior. Any value returned from a settlement handler is ignored so that you can access the original promise's value. Here's an example:

```
const promise = Promise.resolve(42);

promise.finally(() => {
    // settlement handler
    return 43;                  // ignored!
}).then(value => {
    // fulfillment handler
    console.log(value);         // 42
});
```

The `value` passed to the fulfillment handler is `42` and not `43`. The `return` statement in the settlement handler is ignored so that the original value can be retrieved using `then()`. This is one of the consequences of `finally()` returning a promise whose state and value are copied from the original.

Returning Promises in Promise Chains

Returning primitive values from promise handlers allows passing of data between promises, but what if you return an object? If the object is a promise, then there's an extra step that's taken to determine how to proceed. Consider the following example:

```
const promise1 = Promise.resolve(42);
const promise2 = Promise.resolve(43);

promise1.then(value => {
```

```
    console.log(value);     // 42
    return promise2;
}).then(value => {
    console.log(value);     // 43
});
```

In this code, `promise1` resolves to `42`. The fulfillment handler for `promise1` returns `promise2`, a promise already in the resolved state. The second fulfillment handler is called because `promise2` has been fulfilled. If `promise2` were rejected, a rejection handler (if present) would be called instead of the second fulfillment handler.

The important thing to recognize about this pattern is that the second fulfillment handler is not added to `promise2`, but rather to a third promise, making the previous example equivalent to this:

```
const promise1 = Promise.resolve(42);
const promise2 = Promise.resolve(43);

const promise3 = promise1.then(value => {
    console.log(value);     // 42
    return promise2;
});

promise3.then(value => {
    console.log(value);     // 43
});
```

Here, it's clear that the second fulfillment handler is attached to `promise3` rather than `promise2`. This is a subtle but important distinction, as the second fulfillment handler will not be called if `promise2` is rejected. For instance:

```
const promise1 = Promise.resolve(42);
const promise2 = Promise.reject(43);

promise1.then(value => {
    console.log(value);     // 42
    return promise2;
}).then(value => {
```

```
    console.log(value);     // never called
});
```

In this example, the second fulfillment handler is never called because `promise2` is rejected. You could, however, attach a rejection handler instead:

```
const promise1 = Promise.resolve(42);
const promise2 = Promise.reject(43);

promise1.then(value => {
    console.log(value);     // 42
    return promise2;
}).catch(value => {
    console.error(value);   // 43
});
```

Here, the rejection handler is called as a result of `promise2` being rejected. The rejected value `43` from `promise2` is passed into that rejection handler.

Returning a promise from a fulfillment handler is helpful when an operation requires more than one promise to execute to completion. For example, `fetch()` requires a second promise to read the body of a response. To read a JSON body, you'll need to use `response.json()`, which returns another promise. Here's how it looks without using promise chaining:

```
const promise1 = fetch("books.json");

promise1.then(response => {

    promise2 = response.json();
    promise2.then(payload => {
        console.log(payload);
    }).catch(reason => {
        console.error(reason.message);
    });

}).catch(reason => {
    console.error(reason.message);
});
```

This code requires two different rejection handlers to catch the potential errors at two different steps of the process. Returning the second promise from the first fulfillment handler simplifies the code:

```
const promise = fetch("books.json");

promise.then(response => {
    return response.json();
}).then(payload => {
    console.log(payload);
}).catch(reason => {
    console.error(reason.message);
});
```

Here, the first fulfillment handler is called when a response is received and then returns a promise to read the response body as JSON. The second fulfillment handler is called when the body has been read and the payload

> is ready to be used. You need only one rejection handler at the end of the promise chain to catch errors that occur along the way.

Returning a promise from a settlement handler using `finally()` also exhibits some different behavior than using `then()` or `catch()`. First, if you return a fulfilled promise from a settlement handler, then that promise is ignored in favor of the value from the original promise, as in this example:

```
const promise = Promise.resolve(42);

promise.finally(() => {
    return Promise.resolve(44);
}).then(value => {
    console.log(value);     // 42
});
```

In this example, the settlement handler returns a promise that is fulfilled with `44`, but the returned promise is fulfilled with the original promise's value, which is `42`.

However, if you return a rejected promise from a settlement handler, then the returned promise adopts that reason and the returned promise is rejected, like this:

```
const promise = Promise.resolve(42);

promise.finally(() => {
    return Promise.reject(43);
}).catch(reason => {
    console.error(reason);  // 43
});
```

This holds true even if the original promise is rejected, as in this example:

```
const promise = Promise.reject(43);

promise.finally(() => {
    return Promise.reject(45);
}).catch(reason => {
```

```
    console.log(reason);    // 45
});
```

Returning a rejected promise from a settlement handler is functionally equivalent to throwing an error: the returned promise is rejected with the specified reason.

Returning promises from fulfillment or rejection handlers doesn't change when the promise executors are executed. The first defined promise will run its executor first; then the second promise executor will run, and so on. Returning promises simply allows you to define additional responses to the promise results. You defer the execution of fulfillment handlers by creating a new promise within a fulfillment handler. For example:

```
const p1 = Promise.resolve(42);

p1.then(value => {
    console.log(value);     // 42

    // create a new promise
    const p2 = new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(43);
        }, 500);
    });

    return p2;
}).then(value => {
    console.log(value);     // 43
});
```

In this example, a new promise is created within the fulfillment handler for `p1`. That means the second fulfillment handler won't execute until after `p2` is fulfilled. The executor for `p2` resolves the promise after 500 milliseconds using `setTimeout()`, but more realistically you might make a network or file system request. This pattern is useful when you want to wait until a previous promise has been settled before starting a new asynchronous operation.

Summary

Multiple promises can be chained together in a variety of ways to pass information between them. Each call to `then()`, `catch()`, and `finally()` creates and returns a new promise that is resolved when the preceding promise is settled. If the promise handler returns a value, then that value becomes the value of the newly created promise from `then()` and `catch()` (`finally()` ignores this value); if the promise handler throws an error, then the error is caught and the returned newly created promise is rejected using that error as the reason.

When one promise is rejected in a chain, the promises created from other chained handlers are also rejected until the end of the chain is reached. Knowing this, it's recommended to attach a rejection handler at the end of each promise chain to ensure that errors are handled correctly. Failing to catch a promise rejection will result in a message being output to the console, an error being thrown, or both (depending on the runtime environment).

You can return promises from handlers, and in that case, the promise returned from the call to `then()` and `catch()` will settle to match the settlement state and value of the promise returned from the handler (fulfilled promises returned from `finally()` are ignored while rejected promises are honored). You can use this to your advantage by delaying some operations until a promise is fulfilled, then initiating and returning a second promise to continue using the same promise chain.

This chapter explored how to chain multiple promises together so they act more like one promise. In this next chapter, you'll learn how to work with multiple promises acting in parallel.

Final Thoughts

When promises were added into the JavaScript language in 2015, they were a source of controversy and the topic of many thinkpieces opining whether this was the right way to address the asynchronous future of JavaScript. After several years, the dust has settled and promises have won many over, especially with the introduction of async functions in 2017. All new asynchronous JavaScript APIs are built to make use of promises, so understanding how to work with promises is an important part of any JavaScript-focused job.

I hope you've enjoyed this exploration of JavaScript promises.

Purchase the Full Version

You've been reading the community version of this book. If you've enjoyed reading it, please consider purchasing the full version at https://bit.ly/promises-full-ebook. The benefits of the full version include:

1. Four additional chapters:
    1. Responding to Multiple Promises
    2. Async Functions
    3. Abortable Functions
    4. Unhandled Rejection Tracking
2. PDF, Mobi, and ePub formats
3. Free updates

It takes a lot of time and effort to create a book like this, so I'd appreciate your consideration.

Help and Support

If you have any questions or comments about this book, please email books@humanwhocodes.com. Be sure to mention the title of this book in the subject line.

Follow the Author

You can follow Nicholas C. Zakas on the following sites:

- **Blog:** humanwhocodes.com
- **Mastodon:** @nzakas@fosstodon.org
- **X/Twitter:** @slicknet, @humanwhocodes
- **GitHub:** @nzakas, @humanwhocodes
- **Instagram:** @nzakas, @humanwhocodes

Reach out and say hi!