Promises and Asynchronous Programming

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 be able to respond to asynchronous user interactions such as clicks and key presses from the beginning. Node.js further popularized asynchronous programming in JavaScript by using callbacks as an alternative to events. As more and more programs started using asynchronous programming, events and callbacks were no longer powerful enough 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.

To have a good understanding of how promises work, however, it's important to understand some of the basic concepts upon which they are built.

Asynchronous Programming Background

JavaScript engines are built on the concept of a single-threaded event loop. *Single-threaded* means that only one piece of code is ever executed at a time. Contrast this with languages like Java or C++, where threads can allow multiple different pieces of code to execute at the same time. Maintaining and protecting state when multiple pieces of code can access and change that state is a difficult problem and a frequent source of bugs in thread-based software.

JavaScript engines can only execute one piece of code at a time, so they need to keep track of code that is meant to run. That code is kept in a *job queue*. Whenever a piece of code is ready to be executed, it is added to the job queue. When the JavaScript engine is finished executing code, the event loop executes the next job in the queue. The *event loop* is a process inside the JavaScript engine that monitors code execution and manages the job queue. Keep in mind that as a queue, job execution runs from the first job in the queue to the last.

The Event Model

When a user clicks a button or presses a key on the keyboard, an *event* like `onclick` is triggered. That event might respond to the interaction by adding a new job to the back of the job queue. This is JavaScript's most basic form of asynchronous programming. The event handler code doesn't execute until the event fires, and when it does execute, it has the appropriate context. For example:

```js
let button = document.getElementById("my-btn");
button.onclick = function(event) {
    console.log("Clicked");
};
```

In this code, `console.log("Clicked")` will not be executed until `button` is clicked. When `button` is clicked, the function assigned to `onclick` is added to the back of the job queue and will be executed when all other jobs ahead of it are complete.

Events work well for simple interactions, but chaining multiple separate asynchronous calls together is more complicated because you must keep track of the event target (`button` in the previous example) for each event. Additionally, you need to ensure all appropriate event handlers are added before the first time an event occurs. For instance, if `button` were clicked before `onclick` is assigned, nothing would happen. So while events are useful for responding to user interactions and similar infrequent functionality, they aren't very flexible for more complex needs.

The Callback Pattern

When Node.js was created, it advanced the asynchronous programming model by popularizing the callback pattern of programming. The callback pattern is similar to the event model because the asynchronous code doesn't execute until a later point in time. It's different because the function to call is passed in as an argument, as shown here:

```
readFile("example.txt", function(err, contents) {
    if (err) {
        throw err;
    }

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

This example uses the traditional Node.js *error-first* callback style. The `readFile()` function is intended to read from a file on disk (specified as the first argument) and then execute the callback (the second argument) when complete. If there's an error, the `err` argument of the callback is an error object; otherwise, the `contents` argument contains the file contents as a string.

Using the callback pattern, `readFile()` begins executing immediately and pauses when it starts reading from the disk. That means `console.log("Hi!")` is output immediately after `readFile()` is called, before `console.log(contents)` prints anything. When `readFile()` finishes, it adds a new job to the end of the job queue with the callback function and its arguments. That job is then executed upon completion of all other jobs ahead of it.

The callback pattern is more flexible than events because chaining multiple calls together is easier with callbacks. For example:

```
readFile("example.txt", function(err, contents) {
    if (err) {
        throw err;
    }

    writeFile("example.txt", function(err) {
        if (err) {
            throw err;
        }

        console.log("File was written!");
```

```
    });
});
```

In this code, a successful call to `readFile()` results in another asynchronous call, this time to the `writeFile()` function. Note that the same basic pattern of checking `err` is present in both functions. When `readFile()` is complete, it adds a job to the job queue that results in `writeFile()` being called (assuming no errors). Then, `writeFile()` adds a job to the job queue when it finishes.

This pattern works fairly well, but you can quickly find yourself in *callback hell*. Callback hell occurs when you nest too many callbacks, like this:

```
method1(function(err, result) {

    if (err) {
        throw err;
    }

    method2(function(err, result) {

        if (err) {
            throw err;
        }

        method3(function(err, result) {

            if (err) {
                throw err;
            }

            method4(function(err, result) {

                if (err) {
                    throw err;
                }

                method5(result);
            });

        });

    });

});
```

Nesting multiple method calls as this example does creates a tangled web of code that is hard to understand and debug. Callbacks also present problems when you want to implement more complex functionality. What if you want two asynchronous operations to run in parallel and notify you when they're both complete? What if you'd like to start two asynchronous operations at a time but only take the result of the first one to complete?

In these cases, you'd need to track multiple callbacks and cleanup operations, and promises greatly improve such situations.

Promise Basics

A promise is a placeholder for the result of an asynchronous operation. Instead of subscribing to an event or passing a callback to a function, the function can return a promise, like this:

```
// readFile promises to complete at some point in the future
let promise = readFile("example.txt");
```

In this code, `readFile()` doesn't actually start reading the file immediately; that will happen later. Instead, the function returns a promise object representing the asynchronous read 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 the asynchronous operation hasn't completed yet. A pending promise is considered *unsettled*. The promise in the last example is in the pending state as soon as the `readFile()` function returns it. Once the asynchronous operation completes, the promise is considered *settled* and enters one of two possible states:

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

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.

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. Any additional data related to the asynchronous operation is passed to this fulfillment function. The second argument is a function to call when the promise is rejected. Similar to the fulfillment function, the rejection function is passed any additional data related to the rejection.

I> 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:

```
let promise = readFile("example.txt");

promise.then(function(contents) {
    // fulfillment
    console.log(contents);
```

```
}, function(err) {
    // rejection
    console.error(err.message);
});

promise.then(function(contents) {
    // fulfillment
    console.log(contents);
});

promise.then(null, function(err) {
    // rejection
    console.error(err.message);
});
```

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

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:

```
promise.catch(function(err) {
    // rejection
    console.error(err.message);
});

// is the same as:

promise.then(null, function(err) {
    // rejection
    console.error(err.message);
});
```

The intent behind `then()` and `catch()` is for you to use them in combination to properly handle the result of asynchronous operations. This system is better than events and callbacks because it makes whether the operation succeeded or failed 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, all failures will happen silently. Always attach a rejection handler, even if the handler just logs the failure.

A fulfillment or rejection handler will still be executed even if it is added to the job queue 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:

```
let promise = readFile("example.txt");

// original fulfillment handler
```

```
promise.then(function(contents) {
    console.log(contents);

    // now add another
    promise.then(function(contents) {
        console.log(contents);
    });
});
```

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 job queue and called when ready. Rejection handlers work the same way.

I> Each call to `then()` or `catch()` creates a new job to be executed when the promise is resolved. But these jobs end up in a separate job queue that is reserved strictly for promises. The precise details of this second job queue aren't important for understanding how to use promises so long as you understand how job queues work in general.

Creating 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. The `resolve()` function is called when the executor has finished successfully to signal that the promise is ready to be resolved, while the `reject()` function indicates that the executor has failed.

Here's an example that uses a promise in Node.js to implement the `readFile()` function from earlier in this chapter:

```
// Node.js example

let fs = require("fs");

function readFile(filename) {
    return new Promise(function(resolve, reject) {

        // trigger the asynchronous operation
        fs.readFile(filename, { encoding: "utf8" }, function(err,
contents) {

            // check for errors
            if (err) {
                reject(err);
                return;
            }

            // the read succeeded
            resolve(contents);

        });
```

```
    });
}

let promise = readFile("example.txt");

// listen for both fulfillment and rejection
promise.then(function(contents) {
    // fulfillment
    console.log(contents);
}, function(err) {
    // rejection
    console.error(err.message);
});
```

In this example, the native Node.js `fs.readFile()` asynchronous call is wrapped in a promise. The executor either passes the error object to the `reject()` function or passes the file contents to the `resolve()` function.

Keep in mind that the executor runs immediately when `readFile()` is called. When either `resolve()` or `reject()` is called inside the executor, a job is added to the job queue to resolve the promise. This is called *job scheduling*, and if you've ever used the `setTimeout()` or `setInterval()` functions, then you're already familiar with it. In job scheduling, you add a new job to the job queue to say, "Don't execute this right now, but execute it later." For instance, the `setTimeout()` function lets you specify a delay before a job is added to the queue:

```
// add this function to the job queue after 500ms have passed
setTimeout(function() {
    console.log("Timeout");
}, 500);

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

This code schedules a job to be added to the job queue after 500ms. The two `console.log()` calls produce the following output:

```
Hi!
Timeout
```

Thanks to the 500ms delay, the output that the function passed to `setTimeout()` was shown after the output from the `console.log("Hi!")` call.

Promises work similarly. The promise executor executes immediately, before anything that appears after it in the source code. For instance:

```
let promise = new Promise(function(resolve, reject) {
    console.log("Promise");
```

```
    resolve();
});

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

The output for this code is:

```
Promise
Hi!
```

Calling `resolve()` triggers an asynchronous operation. Functions passed to `then()` and `catch()` are executed asynchronously, as these are also added to the job queue. Here's an example:

```
let promise = new Promise(function(resolve, reject) {
    console.log("Promise");
    resolve();
});

promise.then(function() {
    console.log("Resolved.");
});

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

The output for this example is:

```
Promise
Hi!
Resolved
```

Note that even though the call to `then()` appears before the `console.log("Hi!")` line, it doesn't actually execute until later (unlike the executor). That's because fulfillment and rejection handlers are always added to the end of the job queue after the executor has completed.

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 just a single known value, then it doesn't make sense to schedule a job that simply passes a value to the `resolve()` function. Instead, there are two methods that create settled promises given a specific value.

Using Promise.resolve()

The `Promise.resolve()` method accepts a single argument and returns a promise in the fulfilled state. That means no job scheduling occurs, and you need to add one or more fulfillment handlers to the promise to retrieve the value. For example:

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

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

This code creates a fulfilled promise so the fulfillment handler receives 42 as `value`. 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.

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:

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

promise.catch(function(value) {
    console.log(value);         // 42
});
```

Any additional rejection handlers added to this promise would be called, but not fulfillment handlers.

I> 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 called after the `then()` function.

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

```
let thenable = {
    then: function(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:

```
let thenable = {
    then: function(resolve, reject) {
        resolve(42);
    }
};

let p1 = Promise.resolve(thenable);
p1.then(function(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 `p1` is created in the fulfilled state with the value passed from `thenable` (that is, 42), and the fulfillment handler for `p1` receives 42 as the value.

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

```
let thenable = {
    then: function(resolve, reject) {
        reject(42);
    }
};

let p1 = Promise.resolve(thenable);
p1.catch(function(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 `p1`.

`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 ECMAScript 6, 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.

Executor Errors

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

```
let promise = new Promise(function(resolve, reject) {
    throw new Error("Explosion!");
});
```

```
promise.catch(function(error) {
    console.log(error.message);     // "Explosion!"
});
```

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

```
let promise = new Promise(function(resolve, reject) {
    try {
        throw new Error("Explosion!");
    } catch (ex) {
        reject(ex);
    }
});

promise.catch(function(error) {
    console.log(error.message);     // "Explosion!"
});
```

The executor handles catching any thrown errors to simplify this common use case, but an error thrown in the executor is only reported when a rejection handler is present. Otherwise, the error is suppressed. This became a problem for developers early on in the use of promises, and JavaScript environments address it by providing hooks for catching rejected promises.

Global Promise Rejection Handling

One of the most controversial aspects of promises is the silent failure that occurs when a promise is rejected without a rejection handler. Some consider this the biggest flaw in the specification as it's the only part of the JavaScript language that doesn't make errors apparent.

Determining whether a promise rejection was handled isn't straightforward due to the nature of promises. For instance, consider this example:

```
let rejected = Promise.reject(42);

// at this point, rejected is unhandled

// some time later...
rejected.catch(function(value) {
    // now rejected has been handled
    console.log(value);
});
```

You can call `then()` or `catch()` at any point and have them work correctly regardless of whether the promise is settled or not, making it hard to know precisely when a promise is going to be handled. In this case, the promise is rejected immediately but isn't handled until later.

While it's possible that the next version of ECMAScript will address this problem, both browsers and Node.js have implemented changes to address this developer pain point. They aren't part of the ECMAScript 6 specification but are valuable tools when using promises.

Node.js Rejection Handling

In Node.js, there are two events on the `process` object related to promise rejection handling:

- `unhandledRejection`: Emitted when a promise is rejected and no rejection handler is called within one turn of the event loop
- `rejectionHandled`: Emitted when a promise is rejected and a rejection handler is called after one turn of the event loop

These events are designed to work together to help identify promises that are rejected and not handled.

The `unhandledRejection` event handler is passed the rejection reason (frequently an error object) and the promise that was rejected as arguments. The following code shows `unhandledRejection` in action:

```
let rejected;

process.on("unhandledRejection", function(reason, promise) {
    console.log(reason.message);            // "Explosion!"
    console.log(rejected === promise);      // true
});

rejected = Promise.reject(new Error("Explosion!"));
```

This example creates a rejected promise with an error object and listens for the `unhandledRejection` event. The event handler receives the error object as the first argument and the promise as the second.

The `rejectionHandled` event handler has only one argument, which is the promise that was rejected. For example:

```
let rejected;

process.on("rejectionHandled", function(promise) {
    console.log(rejected === promise);              // true
});

rejected = Promise.reject(new Error("Explosion!"));

// wait to add the rejection handler
setTimeout(function() {
    rejected.catch(function(value) {
        console.log(value.message);     // "Explosion!"
    });
}, 1000);
```

Here, the `rejectionHandled` event is emitted when the rejection handler is finally called. If the rejection handler were attached directly to `rejected` after `rejected` is created, then the event wouldn't be emitted. The rejection handler would instead be called during the same turn of the event loop where `rejected` was created, which isn't useful.

To properly track potentially unhandled rejections, use the `rejectionHandled` and `unhandledRejection` events to keep a list of potentially unhandled rejections. Then wait some period of time to inspect the list. For example:

```
let possiblyUnhandledRejections = new Map();

// when a rejection is unhandled, add it to the map
process.on("unhandledRejection", function(reason, promise) {
    possiblyUnhandledRejections.set(promise, reason);
});

process.on("rejectionHandled", function(promise) {
    possiblyUnhandledRejections.delete(promise);
});

setInterval(function() {

    possiblyUnhandledRejections.forEach(function(reason, promise) {
        console.log(reason.message ? reason.message : reason);

        // do something to handle these rejections
        handleRejection(promise, reason);
    });

    possiblyUnhandledRejections.clear();

}, 60000);
```

This is a simple unhandled rejection tracker. It uses a map to store promises and their rejection reasons. Each promise is a key, and the promise's reason is the associated value. Each time `unhandledRejection` is emitted, the promise and its rejection reason are added to the map. Each time `rejectionHandled` is emitted, the handled promise is removed from the map. As a result, `possiblyUnhandledRejections` grows and shrinks as events are called. The `setInterval()` call periodically checks the list of possible unhandled rejections and outputs the information to the console (in reality, you'll probably want to do something else to log or otherwise handle the rejection). A map is used in this example instead of a weak map because you need to inspect the map periodically to see which promises are present, and that's not possible with a weak map.

While this example is specific to Node.js, browsers have implemented a similar mechanism for notifying developers about unhandled rejections.

Browser Rejection Handling

Browsers also emit two events to help identify unhandled rejections. These events are emitted by the `window` object and are effectively the same as their Node.js equivalents:

- `unhandledrejection`: Emitted when a promise is rejected and no rejection handler is called within one turn of the event loop.
- `rejectionhandled`: Emitted when a promise is rejected and a rejection handler is called after one turn of the event loop.

While the Node.js implementation passes individual parameters to the event handler, the event handler for these browser events receives an event object with the following properties:

- `type`: The name of the event (`"unhandledrejection"` or `"rejectionhandled"`).
- `promise`: The promise object that was rejected.
- `reason`: The rejection value from the promise.

The other difference in the browser implementation is that the rejection value (`reason`) is available for both events. For example:

```
let rejected;

window.onunhandledrejection = function(event) {
    console.log(event.type);                    // "unhandledrejection"
    console.log(event.reason.message);          // "Explosion!"
    console.log(rejected === event.promise);    // true
};

window.onrejectionhandled = function(event) {
    console.log(event.type);                    // "rejectionhandled"
    console.log(event.reason.message);          // "Explosion!"
    console.log(rejected === event.promise);    // true
};

rejected = Promise.reject(new Error("Explosion!"));
```

This code assigns both event handlers using the DOM Level 0 notation of `onunhandledrejection` and `onrejectionhandled`. (You can also use `addEventListener("unhandledrejection")` and `addEventListener("rejectionhandled")` if you prefer.) Each event handler receives an event object containing information about the rejected promise. The `type`, `promise`, and `reason` properties are all available in both event handlers.

The code to keep track of unhandled rejections in the browser is very similar to the code for Node.js, too:

```
let possiblyUnhandledRejections = new Map();

// when a rejection is unhandled, add it to the map
window.onunhandledrejection = function(event) {
    possiblyUnhandledRejections.set(event.promise, event.reason);
};

window.onrejectionhandled = function(event) {
    possiblyUnhandledRejections.delete(event.promise);
};
```

```
setInterval(function() {

    possiblyUnhandledRejections.forEach(function(reason, promise) {
        console.log(reason.message ? reason.message : reason);

        // do something to handle these rejections
        handleRejection(promise, reason);
    });

    possiblyUnhandledRejections.clear();

}, 60000);
```

This implementation is almost exactly the same as the Node.js implementation. It uses the same approach of storing promises and their rejection values in a map and then inspecting them later. The only real difference is where the information is retrieved from in the event handlers.

Handling promise rejections can be tricky, but you've just begun to see how powerful promises can really be. It's time to take the next step and chain several promises together.

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()` or `catch()` actually creates and returns another promise. This second promise is resolved only once the first has been fulfilled or rejected. Consider this example:

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

p1.then(function(value) {
    console.log(value);
}).then(function() {
    console.log("Finished");
});
```

The code outputs:

```
42
Finished
```

The call to `p1.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:

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

let p2 = p1.then(function(value) {
    console.log(value);
})

p2.then(function() {
    console.log("Finished");
});
```

In this unchained version of the code, the result of `p1.then()` is stored in `p2`, and then `p2.then()` is called to add the final fulfillment handler. As you might have guessed, the call to `p2.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:

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

p1.then(function(value) {
    throw new Error("Boom!");
}).catch(function(error) {
    console.log(error.message);     // "Boom!"
});
```

In this code, the fulfillment handler for `p1` 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:

```
let p1 = new Promise(function(resolve, reject) {
    throw new Error("Explosion!");
});

p1.catch(function(error) {
    console.log(error.message);     // "Explosion!"
    throw new Error("Boom!");
}).catch(function(error) {
```

```
    console.log(error.message);     // "Boom!"
});
```

Here, the executor throws an error then triggers the p1 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.

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

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:

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

p1.then(function(value) {
    console.log(value);         // "42"
    return value + 1;
}).then(function(value) {
    console.log(value);         // "43"
});
```

The fulfillment handler for `p1` 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:

```
let p1 = new Promise(function(resolve, reject) {
    reject(42);
});

p1.catch(function(value) {
    // first fulfillment handler
    console.log(value);         // "42"
    return value + 1;
}).then(function(value) {
    // second fulfillment handler
    console.log(value);         // "43"
});
```

Here, the executor calls `reject()` with 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.

Returning Promises in Promise Chains

Returning primitive values from fulfillment and rejection 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:

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

let p2 = new Promise(function(resolve, reject) {
    resolve(43);
});

p1.then(function(value) {
    // first fulfillment handler
    console.log(value);     // 42
    return p2;
}).then(function(value) {
    // second fulfillment handler
    console.log(value);     // 43
});
```

In this code, `p1` schedules a job that resolves to 42. The fulfillment handler for `p1` returns `p2`, a promise already in the resolved state. The second fulfillment handler is called because `p2` has been fulfilled. If `p2` 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 `p2`, but rather to a third promise. The second fulfillment handler is therefore attached to that third promise, making the previous example equivalent to this:

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

let p2 = new Promise(function(resolve, reject) {
    resolve(43);
});

let p3 = p1.then(function(value) {
    // first fulfillment handler
    console.log(value);     // 42
    return p2;
});
```

```
p3.then(function(value) {
    // second fulfillment handler
    console.log(value);     // 43
});
```

Here, it's clear that the second fulfillment handler is attached to p3 rather than p2. This is a subtle but important distinction, as the second fulfillment handler will not be called if p2 is rejected. For instance:

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

let p2 = new Promise(function(resolve, reject) {
    reject(43);
});

p1.then(function(value) {
    // first fulfillment handler
    console.log(value);     // 42
    return p2;
}).then(function(value) {
    // second fulfillment handler
    console.log(value);     // never called
});
```

In this example, the second fulfillment handler is never called because p2 is rejected. You could, however, attach a rejection handler instead:

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

let p2 = new Promise(function(resolve, reject) {
    reject(43);
});

p1.then(function(value) {
    // first fulfillment handler
    console.log(value);     // 42
    return p2;
}).catch(function(value) {
    // rejection handler
    console.log(value);     // 43
});
```

Here, the rejection handler is called as a result of p2 being rejected. The rejected value 43 from p2 is passed into that rejection handler.

Returning thenables 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 thenables 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:

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

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

    // create a new promise
    let p2 = new Promise(function(resolve, reject) {
        resolve(43);
    });

    return p2
}).then(function(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. This pattern is useful when you want to wait until a previous promise has settled before triggering another promise.

Responding to Multiple Promises

Up to this point, each example in this chapter has dealt with responding to one promise at a time. Sometimes, however, you'll want to monitor the progress of multiple promises in order to determine the next action. ECMAScript 6 provides two methods that monitor multiple promises: `Promise.all()` and `Promise.race()`.

The Promise.all() Method

The `Promise.all()` method accepts a single argument, which is an iterable (such as an array) of promises to monitor, and returns a promise that is resolved only when every promise in the iterable is resolved. The returned promise is fulfilled when every promise in the iterable is fulfilled, as in this example:

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

let p2 = new Promise(function(resolve, reject) {
    resolve(43);
});

let p3 = new Promise(function(resolve, reject) {
    resolve(44);
```

```
});

let p4 = Promise.all([p1, p2, p3]);

p4.then(function(value) {
    console.log(Array.isArray(value));  // true
    console.log(value[0]);              // 42
    console.log(value[1]);              // 43
    console.log(value[2]);              // 44
});
```

Each promise here resolves with a number. The call to `Promise.all()` creates promise `p4`, which is ultimately fulfilled when promises `p1`, `p2`, and `p3` are fulfilled. The result passed to the fulfillment handler for `p4` is an array containing each resolved value: 42, 43, and 44. The values are stored in the order the promises were passed to `Promise.all`, so you can match promise results to the promises that resolved to them.

If any promise passed to `Promise.all()` is rejected, the returned promise is immediately rejected without waiting for the other promises to complete:

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

let p2 = new Promise(function(resolve, reject) {
    reject(43);
});

let p3 = new Promise(function(resolve, reject) {
    resolve(44);
});

let p4 = Promise.all([p1, p2, p3]);

p4.catch(function(value) {
    console.log(Array.isArray(value))   // false
    console.log(value);                 // 43
});
```

In this example, `p2` is rejected with a value of 43. The rejection handler for `p4` is called immediately without waiting for `p1` or `p3` to finish executing (They do still finish executing; `p4` just doesn't wait.)

The rejection handler always receives a single value rather than an array, and the value is the rejection value from the promise that was rejected. In this case, the rejection handler is passed 43 to reflect the rejection from `p2`.

The Promise.race() Method

The `Promise.race()` method provides a slightly different take on monitoring multiple promises. This method also accepts an iterable of promises to monitor and returns a promise, but the returned promise is settled as soon as the first promise is settled. Instead of waiting for all promises to be fulfilled like the `Promise.all()` method, the `Promise.race()` method returns an appropriate promise as soon as any promise in the array is fulfilled. For example:

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

let p2 = new Promise(function(resolve, reject) {
    resolve(43);
});

let p3 = new Promise(function(resolve, reject) {
    resolve(44);
});

let p4 = Promise.race([p1, p2, p3]);

p4.then(function(value) {
    console.log(value);     // 42
});
```

In this code, `p1` is created as a fulfilled promise while the others schedule jobs. The fulfillment handler for `p4` is then called with the value of 42 and ignores the other promises. The promises passed to `Promise.race()` are truly in a race to see which is settled first. If the first promise to settle is fulfilled, then the returned promise is fulfilled; if the first promise to settle is rejected, then the returned promise is rejected. Here's an example with a rejection:

```
let p1 = new Promise(function(resolve, reject) {
    setTimeout(function() {
        resolve(42);
    }, 100);
});

let p2 = new Promise(function(resolve, reject) {
    reject(43);
});

let p3 = new Promise(function(resolve, reject) {
    setTimeout(function() {
        resolve(44);
    }, 50);
});

let p4 = Promise.race([p1, p2, p3]);

p4.catch(function(value) {
    console.log(value);     // 43
});
```

Here, both `p1` and `p3` use `setTimeout()` (available in both Node.js and web browsers) to delay promise fulfillment. The result is that `p4` is rejected because `p2` is rejected before either `p1` or `p3` is resolved. Even though `p1` and `p3` are eventually fulfilled, those results are ignored because they occur after `p2` is rejected.

Inheriting from Promises

Just like other built-in types, you can use a promise as the base for a derived class. This allows you to define your own variation of promises to extend what built-in promises can do. Suppose, for instance, you'd like to create a promise that can use methods named `success()` and `failure()` in addition to the usual `then()` and `catch()` methods. You could create that promise type as follows:

```
class MyPromise extends Promise {

    // use default constructor

    success(resolve, reject) {
        return this.then(resolve, reject);
    }

    failure(reject) {
        return this.catch(reject);
    }

}

let promise = new MyPromise(function(resolve, reject) {
    resolve(42);
});

promise.success(function(value) {
    console.log(value);             // 42
}).failure(function(value) {
    console.log(value);
});
```

In this example, `MyPromise` is derived from `Promise` and has two additional methods. The `success()` method mimics `then()` and `failure()` mimics the `catch()` method.

Each added method uses `this` to call the method it mimics. The derived promise functions the same as a built-in promise, except now you can call `success()` and `failure()` if you want.

Since static methods are inherited, the `MyPromise.resolve()` method, the `MyPromise.reject()` method, the `MyPromise.race()` method, and the `MyPromise.all()` method are also present on derived promises. The last two methods behave the same as the built-in methods, but the first two are slightly different.

Both `MyPromise.resolve()` and `MyPromise.reject()` will return an instance of `MyPromise` regardless of the value passed because those methods use the `Symbol.species` property (covered under in Chapter 9) to determine the type of promise to return. If a built-in promise is passed to either method, the promise will be resolved or rejected, and the method will return a new `MyPromise` so you can assign fulfillment and rejection handlers. For example:

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

let p2 = MyPromise.resolve(p1);
p2.success(function(value) {
    console.log(value);         // 42
});

console.log(p2 instanceof MyPromise);   // true
```

Here, `p1` is a built-in promise that is passed to the `MyPromise.resolve()` method. The result, `p2`, is an instance of `MyPromise` where the resolved value from `p1` is passed into the fulfillment handler.

If an instance of `MyPromise` is passed to the `MyPromise.resolve()` or `MyPromise.reject()` methods, it will just be returned directly without being resolved. In all other ways these two methods behave the same as `Promise.resolve()` and `Promise.reject()`.

Asynchronous Task Running

In Chapter 8, I introduced generators and showed you how you can use them for asynchronous task running, like this:

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

function run(taskDef) {

    // create the iterator, make available elsewhere
    let task = taskDef();

    // start the task
    let result = task.next();

    // recursive function to keep calling next()
    function step() {

        // if there's more to do
        if (!result.done) {
            if (typeof result.value === "function") {
                result.value(function(err, data) {
                    if (err) {
                        result = task.throw(err);
                        return;
                    }

                    result = task.next(data);
                    step();
```

```
                });
            } else {
                result = task.next(result.value);
                step();
            }

        }
    }

    // start the process
    step();

}

// Define a function to use with the task runner

function readFile(filename) {
    return function(callback) {
        fs.readFile(filename, callback);
    };
}

// Run a task

run(function*() {
    let contents = yield readFile("config.json");
    doSomethingWith(contents);
    console.log("Done");
});
```

There are some pain points to this implementation. First, wrapping every function in a function that returns a function is a bit confusing (even this sentence was confusing). Second, there is no way to distinguish between a function return value intended as a callback for the task runner and a return value that isn't a callback.

With promises, you can greatly simplify and generalize this process by ensuring that each asynchronous operation returns a promise. That common interface means you can greatly simplify asynchronous code. Here's one way you could simplify that task runner:

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

function run(taskDef) {

    // create the iterator
    let task = taskDef();

    // start the task
    let result = task.next();

    // recursive function to iterate through
    (function step() {
```

```
        // if there's more to do
        if (!result.done) {

            // resolve to a promise to make it easy
            let promise = Promise.resolve(result.value);
            promise.then(function(value) {
                result = task.next(value);
                step();
            }).catch(function(error) {
                result = task.throw(error);
                step();
            });
        }
    }());
}

// Define a function to use with the task runner

function readFile(filename) {
    return new Promise(function(resolve, reject) {
        fs.readFile(filename, function(err, contents) {
            if (err) {
                reject(err);
            } else {
                resolve(contents);
            }
        });
    });
}

// Run a task

run(function*() {
    let contents = yield readFile("config.json");
    doSomethingWith(contents);
    console.log("Done");
});
```

In this version of the code, a generic `run()` function executes a generator to create an iterator. It calls `task.next()` to start the task and recursively calls `step()` until the iterator is complete.

Inside the `step()` function, if there's more work to do, then `result.done` is `false`. At that point, `result.value` should be a promise, but `Promise.resolve()` is called just in case the function in question didn't return a promise. (Remember, `Promise.resolve()` just passes through any promise passed in and wraps any non-promise in a promise.) Then, a fulfillment handler is added that retrieves the promise value and passes the value back to the iterator. After that, `result` is assigned to the next yield result before the `step()` function calls itself.

A rejection handler stores any rejection results in an error object. The `task.throw()` method passes that error object back into the iterator, and if an error is caught in the task, `result` is assigned to the next yield result. Finally, `step()` is called inside `catch()` to continue.

This `run()` function can run any generator that uses `yield` to achieve asynchronous code without exposing promises (or callbacks) to the developer. In fact, since the return value of the function call is always coverted into a promise, the function can even return something other than a promise. That means both synchronous and asynchronous methods work correctly when called using `yield`, and you never have to check that the return value is a promise.

The only concern is ensuring that asynchronous functions like `readFile()` return a promise that correctly identifies its state. For Node.js built-in methods, that means you'll have to convert those methods to return promises instead of using callbacks.

A> ### Future Asynchronous Task Running A> A> At the time of my writing, there is ongoing work around bringing a simpler syntax to asynchronous task running in JavaScript. Work is progressing on an `await` syntax that would closely mirror the promise-based example in the preceding section. The basic idea is to use a function marked with `async` instead of a generator and use `await` instead of `yield` when calling a function, such as: A> A> `js A> (async function() { A> let contents = await readFile("config.json"); A> doSomethingWith(contents); A> console.log("Done"); A> })(); A>` A> A> The `async` keyword before `function` indicates that the function is meant to run in an asynchronous manner. The `await` keyword signals that the function call to `readFile("config.json")` should return a promise, and if it doesn't, the response should be wrapped in a promise. Just as with the implementation of `run()` in the preceding section, `await` will throw an error if the promise is rejected and otherwise return the value from the promise. The end result is that you get to write asynchronous code as if it were synchronous without the overhead of managing an iterator-based state machine. A> A> The `await` syntax is expected to be finalized in ECMAScript 2017 (ECMAScript 8).

Summary

Promises are designed to improve asynchronous programming in JavaScript by giving you more control and composability over asynchronous operations than events and callbacks can. Promises schedule jobs to be added to the JavaScript engine's job queue for execution later, while a second job queue tracks promise fulfillment and rejection handlers to ensure proper execution.

Promises have three states: pending, fulfilled, and rejected. A promise starts in a pending state and becomes fulfilled on a successful execution or rejected on a failure. 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 and the `catch()` method allows you to assign only a rejection handler.

You can chain promises together in a variety of ways and pass information between them. Each call to `then()` creates and returns a new promise that is resolved when the previous one is resolved. Such chains can be used to trigger responses to a series of asynchronous events. You can also use `Promise.race()` and `Promise.all()` to monitor the progress of multiple promises and respond accordingly.

Asynchronous task running is easier when you combine generators and promises, as promises give a common interface that asynchronous operations can return. You can then use generators and the `yield` operator to wait for asynchronous responses and respond appropriately.

Most new web APIs are being built on top of promises, and you can expect many more to follow suit in the future.