Programming JavaScript

Santosh Kalwar, 15.02.2022

JavaScript Topics

- Asynchronous callbacks

- Practice

- Callback pattern

- Intro to Promises
    - Promises vs. Callbacks
    - Introduction to promises
    - Practice

- Using promises
    - visualize promise states
    - resolving data
    - creating and rejecting promises
    - Practice

Mini projects
    - Miniproject01
    - Miniproject02

SetTimeout

The next set of chapters will deal with asynchronous JavaScript. What does the word asynchronous mean? And what is asynchronous JavaScript?

The goal of the next few chapters is to be able to integrate with other services from the Internet. For example, integrate with a weather API (which allows getting the weather forecast in a certain location).

Let's take the example of the weather forecast. When you ask a certain service/API for the weather forecast in a certain location, the result will *not* be instantaneous. It will take anywhere between 100ms and 1s (on average). That's because your application has to send a request to another server, which then has to process it and then finally send you the response back.

What we mean by that, is that these things often take some time. On the other hand, the following code executes immediately:

```
console.log("A");
console.log("B");
```

You will see `A` and then immediately `B` followed by it.

But, that's not the case with working with APIs. And, in order to simplify the topics, we will not work with an API just yet, we will start by simulating a result that takes some time to complete.

setTimeout

To do that, we'll use a browser function called `setTimeout`. Here's how it works:

```
setTimeout(() => {
    console.log("One second has elapsed.");
}, 1000);
```

The function signature for `setTimeout` is `setTimeout(callback, milliseconds)`.

When you run the code above, you will see `"One second has elapsed."` approximately 1 second after you run the code.

So, what the `setTimeout` function does is that it **queues** the `callback` function that you specify in the future. It will wait the `milliseconds` that you specified. In this example, we specified 1000 milliseconds which is the equivalent of 1 second.

setTimeout

The callback is the following function:

```
() => {
    console.log("One second has elapsed.");
}
```

This callback is similar to the previous callbacks that we saw in the array chapters, except that this one is running somewhere in the future. In this example, 1 second in the future. Because it runs somewhere in the future, we call it an **asynchronous callback**.

Execution order

Your JavaScript code **always** runs top to bottom. However, some parts of the code might be queued for the future.

This can be confusing to beginner developers and experienced developers coming from back-end programming languages. But don't worry, the concept can be explained and understood.

Let's take a look at this example:

```
console.log("A");
setTimeout(() => {
    console.log("B")
}, 1000);
console.log("C");
```

What do you think the output of this code is?

Execution order

Take your time and think about it, knowing the following: Your JavaScript code **always** runs top to bottom. However, some parts of the code might be queued for the future.

The output of this code is the following:

```
"A"
"C"
"B" // this one shows up 1 second later
```

Let's break down the execution order of the code above, knowing that your code is running top to bottom:

1. `console.log("A");` => this will immediately log `"A"` to the console.
2. The next line starts at `setTimeout(() => {` and finishes at `}, 1000);`.
   The full line is:

```
setTimeout(() => {
    console.log("B")
}, 1000);
```

This line of code runs immediately, however, as we learned in the previous lesson, the `setTimeout` function will **queue** the **callback** into the future.
This means that `() => { console.log("B") }` is now queued 1 second into the future, and will not run immediately. Thus, the 2nd line of code does not log anything just yet.

3. The last line of code is `console.log("C");` which immediately logs `"C"` to the console.

What do synchronous and asynchronous mean?

To explain the difference between synchronous and asynchronous code, think about the difference between a face-to-face conversation and a conversation over text message.

A face-to-face conversation is synchronous because people are talking and replying immediately one after the other.

However, a conversation over text message is asynchronous because the recipients can reply at a later time in the future.

Chapter recap

- When working with APIs (for example a service that returns the weather forecast), the result/response will take some time to come back. This is why we're learning about asynchronous JavaScript.

- `setTimeout(callback, milliseconds)` is a JavaScript function that queues the `callback` into the future (depending on the `milliseconds` you specify).

- An asynchronous callback is a callback that runs somewhere in the future.

- Your JavaScript code **always** runs top to bottom. However, some parts of the code might be queued for the future.

- A face-to-face conversation is synchronous because people are talking and replying immediately one after the other.

- However, a conversation over text message is asynchronous because the recipients can reply at a later time in the future.

Callback pattern

Before we learn about this technique, please note that there's a better way of doing it using promises. I am introducing this technique here so that the idea of promises in the next chapter makes more sense, also, because you might still see this pattern used in old browser functions and some old JavaScript libraries.

```
welcomeUser("Sam", () => {
    console.log("Done welcoming user");
});
```

The example above assumes a function `welcomeUser` that accepts the `name` of the user as its first parameter, followed by the *success callback*.

The success **callback** is an asynchronous callback that will run when the function has finished its tasks.
Note that the second parameter `() => {console.log("Done welcoming user")}` is a function definition. The implementation of `welcomeUser` is the one saying that we're going to call this function once we're done welcoming the user.

Callback pattern

The code for `welcomeUser` would look like this, we've added a `setTimeout` to simulate an action in the future:

```
const welcomeUser = (name, callback) => {
    setTimeout(() => {
        console.log(`Welcome ${name}`);
        callback(); // call the success callback function
    }, 1000);
}
```

Don't worry about the syntax for `welcomeUser`, we will cover it later on and see how you can pass functions around. What we care about for now is the coding pattern where you can pass a function as a parameter and that function will be called when the function has finished executing (most of the time).

Why do callbacks exist?

The reason why callbacks exist and why it's possible in JavaScript to schedule work for the future is *performance*.

We cannot block the execution of JavaScript until a certain long task has finished. Imagine if, while the `welcomeUser` is executing, we had to wait 1 second. Meaning that the browser becomes completely unusable for a full second until it has finished.

This is the reason why we have callbacks (and later promises). It allows us to continue running the rest of the code while scheduling some functions in the future.

Callback data & handling errors

The success callback is able to receive some `data`. For example, let's say there's an expensive calculation happening and you'd like to make it possible for the success callback to receive the answer of that calculation. Here's an example:

```
const temperatures = [10, 5, 3];

sumTemperatures(temperatures, value => {
    console.log(value); // 18 (the sum of temperatures)
});
```

The result will be `18` logged to the console after 1 second.

Callback data & handling errors

The code above assumes the following implementation for `sumTemperatures`:

```
const sumTemperatures = (temperatures, callback) => {
    // setTimeout to simulate an "expensive calculation"
    setTimeout(() => {
        const sum = temperatures.reduce((total, current) => total + current, 0);
        callback(sum); // call the success callback (with the result of the sum)
    }, 1000);
}
```

So the callback:

```
value => {
    console.log(value);
}
```

is being called automatically by the `sumTemperatures` function and it is given the sum as the first parameter. So, for you to read that result of the sum, you just have to define the first parameter and name it whatever you want. In this example, we called it `value`.

Now, `value` is available inside this function (within the `{ ... }`).

Handling errors

Things get even more complicated when you want to handle errors. Some functions might break, for example, if you're trying to get the weather from an API/service, this might break because of a bad Internet connection, or maybe the weather service is down, etc.

Using the callback pattern, we often see the *error callback* specified after the *success callback*. For example:

```
sumTemperatures(temperatures, value => {
    console.log(value); // 18 (the sum of temperatures)
}, reason => {
    // this callback will run when there's an error
    console.error(reason);
});
```

The function signature of sumTemperatures becomes:

```
sumTemperatures(temperatures, successCallback, errorCallback);
```

Handling errors

Things get even more complicated when you want to handle errors. Some functions might break, for example, if you're trying to get the weather from an API/service, this might break because of a bad Internet connection, or maybe the weather service is down, etc.

Using the callback pattern, we often see the *error callback* specified after the *success callback*. For example:

```
sumTemperatures(temperatures, value => {
    console.log(value); // 18 (the sum of temperatures)
}, reason => {
    // this callback will run when there's an error
    console.error(reason);
});
```

The function signature of `sumTemperatures` becomes:

```
sumTemperatures(temperatures, successCallback, errorCallback);
```

One of the annoying things about this pattern is that the only differentiation between the success callback and the error callback is their order in the parameters. This is not very clear and can lead to confusion.

There's one more annoying result of the callback pattern called callback hell.

Callback hell

Callbacks can get really tough to read, especially once you start nesting them. Take a look at this example:

```
showLoader(() => {
    getWeather((data) => {
        // success callback
        hideLoader(() => {
            enableRefreshButton();
            displayWeather(data, () => {
                logToAnalytics("weather");
            });
        });
    }, () => {
        // error callback
        hideLoader(() => {
            enableRefreshButton();
        });
    })
});
```

This is what we call *callback hell*. It's when the code becomes too complicated to read because it's too nested. Notice how the `logToAnalytics("weather");` call is nested 4 levels deep.

Also, handling errors is not very elegant. In the next chapter, we'll learn how to consume/use promises!

Practice / Classroom coding

async01.js: Visualize setTimeout, Run the code and notice how the console.log shows up after 2 seconds. Feel free to play around with the code (`callback` and `milliseconds`) and see how that affects the result.

async02.js: Update the `delayedWelcome` function such that it delays the `console.log` call 1 second into the future.

async03.js: Run the code by adding a comment at the end of the file (code changes will re-run the code) and take a look at the output. Feel free to play around with the code and see how that affects the result.

async04.js: Can you guess the output before running the code?

callback01.js: Update the `sayHello` function such that the `console.log("Done!")` runs **after** the `welcomeUser` function has finished executing.

callback02.js: Update the `calculateSum` function such that the `console.log("The sum is: X")` runs **after** the `sumGrades` function has finished executing. Also, replace `X` with the actual sum of the grades.

Chapter Recap

- The callback pattern is a programming pattern where you pass a function definition as a parameter to a function. That callback will then be automatically called once the function call has been completed successfully.
- We have callbacks (and later promises) as they allow the browser to continue responding to user input instead of blocking and waiting until a function has finished executing.
- We're only learning about the callback pattern because we'd like to replace it with promises in the next chapter.
- The success callback is able to receive some data. For example, let's say there's an expensive calculation happening and you'd like to make it possible for the success callback to receive the answer of that calculation.
- One of the annoying things about the callback pattern is that the only differentiation between the success callback and the error callback is their order in the parameters. This is not very clear and can lead to confusion.

Promises vs Callbacks

We saw in the previous section the callback pattern isn't very elegant. This is why promises were created. We'll spend some time working with promises first, and then we'll learn how to create your own promises.

Before I explain promises, let's take a look at a before/after example. Here's an example written with the callback pattern:

```
const temperatures = [10, 5, 3];

sumTemperatures(temperatures, value => {
    console.log(value); // 18 (the sum of temperatures)
});
```

and then refactored to use Promises (this would require a different implementation for `sumTemperatures` but don't worry about it for now):

```
const temperatures = [10, 5, 3];

sumTemperatures(temperatures).then(value => {
    console.log(value); // 18 (the sum of temperatures)
});
```

Notice how we call `sumTemperatures(temperatures)` and then we chain on it `.then(callback)`.
It is now much clearer that the `value => { console.log(value); }` will run **when** the `sumTemperatures(temperatures)` has completed.

We haven't explained promises yet, so don't worry about the `.then()` for now, but only look at the syntax before and after.

Promises vs Callbacks

It gets even better when you want to handle errors. Here's the before/after example:

```
sumTemperatures(temperatures, value => {
    console.log(value); // 18 (the sum of temperatures)
}, reason => {
    // this callback will run when there's an error
    console.error(reason);
});
```

And then refactored to Promises:

```
sumTemperatures(temperatures)
    .then(value => {
        console.log(value); // 18 (the sum of temperatures)
    })
    .catch(reason => {
        // this callback will run when there's an error
        console.error(reason);
    });
```

Do you see how we have 2 callbacks, one for success (`.then()`) and another one for errors (`.catch()`)?.

This is clearer because we don't have to rely on the order of the callbacks, instead, they are scheduled in the future with `.then()` and `.catch()` and one of them will run depending on the outcome of the promise.

Execution order

In the next sections, we'll only be focusing on the `.then()`.

The execution order might still throw you off, however, it's similar to what you learned before:

Your JavaScript code **always** runs top to bottom. However, some parts of the code might be queued for the future.

Assuming a function `wait(milliseconds)` that returns a promise:

```
console.log("A");
wait(1000).then(() => {
    console.log("B");
});
console.log("C");
```

What do you think the output of this code is?

The output is:

```
"A"
"C"
"B" // 1 second later
```

That's because the following line: `wait(1000).then(() => { console.log("B"); });` schedules the `.then()` callback 1 second into the future. So, the callback `() => { console.log("B"); }` will run after 1 second.

Promise on MDN

Intro to promises

Promises is a JavaScript feature that allows us to schedule work in the future and then runs callbacks based on the outcome of the promise (whether it was successful or not). Assuming a function `wait` with the below implementation:

```
// the code below (how to create promises) will be explained at a later chapter
// we only care about consuming/using promises for now.
const wait = milliseconds => {
    // this function returns a new promise
    return new Promise(resolve => {
        setTimeout(() => {
            resolve();
        }, milliseconds);
    });
}
```

Here's how you can use it:

```
wait(1000).then(() => {
    console.log("waited 1 second");
});
```

Because the function `wait()` returns a new promise (as seen in the first code block), then we can call `.then()` on its result. You can only call `.then()` on functions that return a new Promise.

Several Web APIs (functions in the browser) work with promises, which is why this topic is important. Most importantly:

- `fetch` (explained in a later chapter)
- `getUserMedia` (access camera/audio)

Common mistakes

The following code will **not** work:

```
wait(1000); /* ❌ */
.then(() => {
    console.log("This breaks 👎");
})
```

That's because the `.then()` needs to be called on the result of `wait(1000)`. So, by adding a `semi-colon` after `wait(1000)`, the `.then()` is now being called on nothing which breaks.

Another common mistake has to do with trying to read the data resolved by the promise in a synchronous fashion.

Promise.then on MDN

Chapter recap

- Promises are cleaner because callbacks are clearly marked as successful (with `.then()`) and erroneous (with `.catch()`).

- The `.then(callback)` schedules the `callback` into the future when the promise complete successfully (more on that in the next lesson).

- Your JavaScript code **always** runs top to bottom. However, some parts of the code might be queued for the future.

- When a function returns a promise, you can call `.then(callback)` on its result. The callback will be scheduled in the future when the promise completes successfully.

- The `.then()` has to be chained on the result of the function that returns a promise, so, you should not add a `;` in between these two.

Promise states

A promise can have 3 states:

- pending
- fulfilled
- rejected

We'll cover the rejected state in the next section.

When you create a promise, it will start in the pending state. When it has been completed successfully, then it becomes in the fulfilled state.

So, when the promise becomes fulfilled, the callback passed to .then(callback) will execute.

Visualizing the pending state

Assuming a function `wait(milliseconds)` that returns a promise. Let's try to console.log() the result of `wait(1000)`:

```
console.log(wait(1000)); // Promise {<pending>}
```

You can open your browser's dev tools console (Right-click > Inspect element to visualize the state)

The promise shows `pending` which is the state of the promise when it's created. This would also be the case if you save `wait(1000)` in a variable and console.log that variable:

```
const result = wait(1000);
console.log(result); // Promise {<pending>}
```

Now, if you wait 1000 milliseconds (or more) and then try to log `console.log(result)`, you will get `Promise {<fulfilled>: undefined}`. We'll talk about the `undefined` here in the next lesson. Notice how the promise now has a status of `fulfilled`. This means that it has been completed successfully.

Visualizing the pending state

Let's take a look at another example, for pedagogical purposes only:

```
const result = wait(1000);
console.log(result); // Promise {<pending>}
result.then(() => {
    console.log(result); // Promise {<fulfilled: undefined}
});
console.log(result); // Promise {<pending>} (because your code runs top to bottom. However, the promise callback gets scheduled into the future)
```

Notice how the `result` starts as `pending` and then becomes `fulfilled` inside the `.then()` callback.

This is what promises are all about. They let you run a callback sometime in the future when the promise has been completed successfully. (In the next chapter, we'll be able to handle errors too.)

You may have also noticed that the last line `console.log(result)` shows `pending`, that's because your code is running top to bottom. At the time that this line runs, the promise has not completed yet.

Resolving data

When a promise completes successfully, you will often hear/read people say that it has **resolved**. The reason for this naming will become clearer once we learn about the `Promise.resolve()` method.

One of the most useful features of promises is when they resolve with some data, an answer of some computation, or some values coming from an external service/API.

Let's start with the callback equivalent:

```
const temperatures = [10, 5, 3];

sumTemperatures(temperatures, value => {
    console.log(value); // 18 (the sum of temperatures)
});
```

The callback function receives a first argument (that we called `value`, but you can call it whatever you want). This argument represents the result of the computation made by `sumTemperatures`.

Resolving data

The same example above can be re-written with promises (assuming we also rewrite the implementation of sumTemperatures):

```
const temperatures = [10, 5, 3];

sumTemperatures(temperatures).then(value => {
    console.log(value); // 18 (the sum of temperatures)
});
```

The .then() callback receives an argument which is the result of the implementation. You can call it whatever you want, and that will be the variable that you can use inside the callback. For example, the above example can be re-written as:

```
const temperatures = [10, 5, 3];

sumTemperatures(temperatures).then(data => {
    console.log(data); // 18 (the sum of temperatures)
});
```

This is called a promise *resolving data*. This means that the promise is giving us an answer after it has been completed. This will be especially important when we work with fetch.

Resolving data

Some promises will resolve with data (for example, `fetch`), others will resolve with nothing (`undefined`).

If we were to log the promise created by `sumTemperatures()`, here's what you'll get:

```
const sumPromise = sumTemperatures(temperatures);
console.log(sumPromise); // Promise {<pending>}
sumPromise.then(data => {
    console.log(sumPromise); // Promise {<fulfilled: 18>}
    console.log(data); // 18 (the sum of temperatures)
});

console.log(sumPromise); // Promise {<pending>}
```

Notice how you can see the data resolved by the promise next to `fulfilled`. Note that the console in this app might not replicate this behavior accurately.

Use case

To better understand why this is useful, let's take a look at the sample "pseudo-code" (this won't execute successfully, it's just meant to illustrate the purpose of resolving data):

```
getWeatherIn(“Helsinki").then(data => {
    console.log(data); // {degreesC: 10, degreesF: 50, description:
"cloudy"}
    console.log(data.degreesC); // 10
});

getWeatherIn("Tokyo").then(data => {
    console.log(data); // {degreesC: 25, degreesF: 77, description:
"sunny"}
    console.log(data.degreesC); // 25
});
```

Notice how the `getWeatherIn` function resolves with different data, depending on the city that it receives.

Extracting value out of promise (common mistake)

It's a common mistake to try and extract the `data` or `value` (or whatever the promise is resolving) outside of the promise as follows:

```
/* ❌ this does NOT work as expected */
const data = getWeatherIn(“Helsinki");

console.log(data); // Promise <pending>
```

This is **not** possible. You **have to** add a .then() callback and you will *only* be able to access the `data` inside the `.then` callback. This is because the promise callback will only run in the future (once the promise has been completed).

If JavaScript were to support this syntax, then the user would not be able to interact with the browser for the entire duration that `getWeatherIn` needs to complete. If that were the case, no one would use browsers because they would be terribly slow.

There's one way to make the code above work, which is with `async/await` that is explained later chapter. However, it is **very important** to understand promises first because `async/await` is just syntactic sugar on top of promises. This is why we are focusing a lot on promises before explaining `async/await`, and you should follow that too.

Chapter recap

- A promise can have 3 states: `pending`, `fulfilled`, and `rejected`.
- Every promise starts with the `pending` state and then becomes `fulfilled` when it has been completed successfully.
- Promises let you run a callback sometime in the future when the promise has been completed successfully.
- When a promise completes successfully, you will often hear/read people say that it has **resolved**.
- One of the most useful features of promises is when they resolve with some data, an answer of some computation or some values coming from an external service/API.
- You are not able to extract the value resolved by a promise outside of a promise.
- Don't rush into learning `async/await` yet. You have to have solid experience in promises first.

Rejected state

In the previous chapter, we mentioned that a promise can have 3 states:

- `pending`
- `fulfilled`
- `rejected`

Things don't always go as planned. Sometimes, things fail. This is why we have the `rejected` state.

For example, let's say you're trying to connect to a weather service/API, this might fail because of a network connection issue. In that case, the promise will **not** resolve successfully (it will **not** be fulfilled). Instead, it will be `rejected`.

This lets us know that something went wrong which allows us to handle the error by showing an error message or a retry button for example.

In this case, we say that the promise *rejected*. The reason for this naming will become clearer once we learn about the `Promise.reject()` method.

.catch (callback)

The `.catch(callback)` allows you to handle the `rejected` state of a promise. Here's an example:

```
getWeatherIn(“Helsinki")
    .then(data => {
        console.log(data);
    })
    .catch(error => {
        console.error(error); // {error: "Connection issue"}
    });
```

The `.catch(callback)` will run when the promise is not successful.

Just like how the promise can resolve with data when successful, it can also reject with some data when it's not successful.

The first argument (which we called `error` here), can be a string, an object, etc. This will be given to you in the documentation.

Promise.catch on MDN

Promise life cycle

A promise starts in the pending state and then will either resolve successfully (fulfilled) or reject with an error (rejected).

It is also possible for a promise to never resolve. This can be caused by one of 2 reasons:

- wrong implementation of the promise (that should be fixed)
- the condition that resolves the promise is never achieved (this is okay)

An example of the latter, let's say we've got a promise that resolves when the user clicks on a button. If the user never clicks on the button, then the promise will never resolve (or reject). It will always stay in the pending state.

Promise.finally

Before we learn about Promise.finally, let's talk about try...catch with promises.

The try...catch statement does **not** work with promises. That's because the promises are asynchronous meaning that they are happening at a later stage.

So, if a promise fails, this is going to be sometime in the future. And, by that time, the try...catch statement has already been completed a long time ago.

Also, you don't need try...catch because promises have .then() and .catch() that act similarly to the try...catch statement.

Promise.finally

We've learned about .then(callback) and .catch(callback) but there's one more which is .finally(callback). The .finally() callback will execute whenever the promise's state changes from pending to either fulfilled or rejected.
This means that .finally() will execute for both success and error states. Let's take an example:

```
getWeatherIn(“Helsinki")
    .then(data => {
        console.log(data);
        console.log("Done fetching weather");
    })
    .catch(error => {
        console.error(error);
        console.log("Done fetching weather");
    });
```

The above code logs Done fetching weather in both scenarios (success and error). We can refactor it to avoid duplication with .finally():

Promise.finally

```
getWeatherIn(“Helsinki")
    .then(data => {
        console.log(data);
    })
    .catch(error => {
        console.error(error);
    })
    .finally(() => {
        console.log("Done fetching weather");
    });
```

This code does exactly the same thing as the one above but is more elegant as it avoids duplication. The .finally(callback) will execute after the .then() when the promise resolves successfully and after the .catch() when the promise rejects.

Manually rejecting a promise

It's also possible to manually reject a promise by throwing an error yourself. Here's how:

```
getWeatherIn("Helsinki")
    .then(data => {
        throw new Error("Stopped.");
        console.log(data);
        console.log("Done fetching weather");
    })
    .catch(error => {
        console.error(error);
        console.log("Done fetching weather");
    });
```

Assuming the `getWeatherIn()` fulfilled, the `.then()` callback will execute. However, the `throw new Error("Stopped.")` will immediately **reject** the promise.

So, the rest of the `.then` callback is ignored, and the `.catch()` callback will execute.

We will use this pattern to handle fetch calls when the API returns an error.

Useful with DOM and fetch

For now, assume we want to show a loader when we start getting the weather and then stop it afterward. Assuming the functions `startLoader()` and `stopLoader()`, here's how the code would look like before using `.finally()`:

```
startLoader();
getWeatherIn(“Helsinki")
    .then(data => {
        stopLoader();
        console.log(data);
    })
    .catch(error => {
        stopLoader();
        console.error(error);
    });
```

Promise.finally on MDN

And here's how it looks like after refactoring it with `.finally()`:

```
startLoader();
getWeatherIn(“Helsinki")
    .then(data => {
        console.log(data);
    })
    .catch(error => {
        console.error(error);
    })
    .finally(() => {
        stopLoader();
    });
```

Chapter recap

- The rejected state of a promise is meant for when things break. For example, a network connection issue.
- The callback provided to .catch(callback) will run if the promise ends up in the rejected state.
- A promise starts in the pending state and then will either resolve successfully (fulfilled) or reject with an error (rejected).
- It is also possible for a promise to never resolve (could be an implementation bug or could be because the condition to resolve is never met).
- try...catch does not work with promises because the promises are asynchronous.
- Use .catch() to handle errors with promises.
- The .finally(callback) runs after the .then() when the promise resolves successfully and after the .catch() when the promise rejects.
- .finally(callback) can be used to stop a loader in both cases (success and error).

Creating promises

In the previous promise chapters, we used functions that return a promise. In this section, you'll learn how to write a function that returns a promise.

The use cases for that are a bit on the advanced side. Most of the time you will be consuming (using) functions that return a promise (like the `fetch` API that we will learn about soon). Also, you will benefit by better understanding how promises work.

Let's see how we can implement the `waitOneSecond` function that resolves successfully after 1 second. We start by defining the function:

```
const waitOneSecond = () => {

}
```

Now, this function must return a new promise:

```
const waitOneSecond = () => {
    return new Promise(() => {

    });
}
```

This means that the `waitOneSecond` function is doing some asynchronous work. So, its result will not be directly accessible because it has to resolve somewhere in the future. If you forget to `return new Promise(...)` then you won't be able to call `.then()` or `.catch()` on the result of `waitOneSecond()`.

Creating promises

Notice the syntax `new Promise()`. The `Promise` class is available in the JavaScript language and you need to create a new instance of it.

This `Promise` class receives one argument which is called the `executor` (`() => {}`). The executor will run immediately.

Also, this `executor` receives as the first argument a function that you can call whenever the promise has completed its work. Even though you can call this argument whatever you want, it is *very common* to call it `resolve`. Here's how it looks like:

```
const waitOneSecond = () => {
    return new Promise((resolve) => {
        // do some work
        // when it's done, call resolve()
    });
}
```

Let's say we want this function to resolve asap, then you can call the `resolve()` method directly inside of the promise executor:

```
const waitOneSecond = () => {
    return new Promise((resolve) => {
        resolve();
    });
}
```

Creating promises

However, for our use case, we need to wait 1 second before resolving, thus the code will be:

```
const waitOneSecond = () => {
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve();
        }, 1000);
    });
}
```

When you call `resolve()`, the promise state will be changed from `pending` to `fulfilled`. This is why when you call `resolve()`, then `.then()` callback will execute.

Here's how you can use the `waitOneSecond` function:

```
waitOneSecond().then(() => {
    console.log("Waited one second");
});
```

Resolving data

Two chapters ago we saw how a promise can resolve with data. For example, when you ask for weather data, the promise will resolve with weather data. In this lesson, we'll learn how this works.

Let's make our previous `wait()` function resolve with the number of **seconds** it waited (it still receives `milliseconds` but it resolves with the number of **seconds**):

```
const wait = milliseconds => {
    return new Promise((resolve) => {
        setTimeout(() => {
            const seconds = milliseconds / 1000;
            resolve(seconds);
        }, milliseconds);
    });
}
```

Notice how the `resolve()` method can receive an argument. This argument will then be available in the `.then(callback)`.

Resolving data

So, the code above can be used as following:

```
wait(2000).then(data => {
    console.log(data); // 2 (seconds)
});
```

You can, as usual, rename `data` into `seconds` or whatever variable name you want:

```
wait(2000).then(seconds => {
    console.log(seconds); // 2 (seconds)
});
```

Chapter recap

- When implementing a function that returns a promise, you have to `return new Promise(() => {})`.
- The `() => {}` function inside the Promise constructor is called `executor`.
- The executor receives as the first argument the `resolve` function. You should call the `resolve()` function when the promise needs to move from `pending` to `fulfilled`.
- Calling `resolve()` with a certain value will make that value available in the `.then(callback)`.

Rejecting promises

In the previous chapter, you learned how to define your function that returns a promise and how to resolve it successfully.
In this lesson, we'll learn how to reject it with an error (when necessary).

The promise executor function (the function passed to the `new Promise(...)` call), in fact, receives 2 arguments. The first one we called `resolve` and we saw in the last lesson. The second one is the function that you can call to move the promise from `pending` to `rejected`. We will call this one `reject`.

Thus, here's how promises will look like:

```
new Promise((resolve, reject) => {
    resolve(); // will resolve the promise (.then(callback) will be
called)

    reject(); // will fail the promise (.catch(callback) will be called)
});
```

Note that the example above is not complete, you still have to decide when to call `resolve()` or when to call `reject()` (based on some condition for example).

Rejecting with data

The `reject()` can also pass some data to the `.catch()` callback. For example:

```
const alwaysFail = () => {
    return new Promise((resolve, reject) => {
        reject("Failed. That's the only thing I do.")
    });
}

alwaysFail
  .then(() => {
    // will never be called
  })
  .catch(data => {
    console.error(data); // "Failed. That's the only thing I do."
  });
```

Practice / Classroom coding

promise01.js: Run the code by adding a comment at the end of the file (code changes will re-run the code) and take a look at the output.

promise02.js: The function `randomWait` waits between 1 and 4 seconds (it chooses a random value every time). Run the code and take a look at the output.

promise03.js: Fix the code in the `init` function such that the 2nd console.log (`"Waited 1 second"`) logs after `wait(1000)` has finished executing.

promise04.js: Call this fakeFetch function inside the init function and then move the console.log("Fake fetch completed") such that it runs once fakeFetch has completed.

promise05.js: Run the code and take a look at the output. Visualize the promise state at every step (using your browser's dev tools console). Make sure you understand why the last line logs `pending` rather than `fulfilled`.

promise06.js: Update the code in `index.js` such that the `console.log("Complete clicked");` runs after you click on the complete button.

Practice / Classroom coding

promise07.js: Implement the `logWeatherDescription` such that it uses `getWeatherDescription` to log the weather description to the console.Look at the sample usage to see the expected result.

promise08.js: Call this `fakeFetch` function inside the `logFlightStatus` function and then log the data that it receives once it has completed.

promise09.js: This will fail at the moment since it's not implemented so make sure to handle the rejected case.To handle the rejected case, you need to log the error using `console.error`.

promise10.js: Update the code in `index.js` such that the `console.log("Complete clicked");` runs after you click on the complete button and `console.error("Fail clicked")` runs after you click on the fail button.

promise11.js: Update the code getting rid of the `stopLoader()` duplication (it should only be called once).

Practice / Classroom coding

promise12.js: We've implemented most of the `waitOneSecond` function. However, it's always in the `pending` state. Fix the implementation so that it moves to the `fulfilled` state after 1 second.

promise13.js: Implement a function `wait(milliseconds)` that returns a promise and fulfills after `milliseconds` have elapsed. Look at the sample usage to see how the function is being used.

promise14.js: Complete the `fakeFetch` function such that it returns a promise that resolves successfully after 1 second with the following object

promise15.js: Complete the fakeFetch(endpoint) function

promise16.js: Implement the function `failAfter(milliseconds)` that returns a promise and fails after `milliseconds` have elapsed. It should fail with the following message: `"You asked me to fail after Xms and I did!"` where `X` is replaced by `milliseconds`.

Chapter recap

- The 2nd argument in the promise executor is a function that you can call to reject the promise. It will move it from `pending` to `rejected`.

- Even though you're not required, aim to call this 2nd argument `reject` for clarity reasons.

- In summary, the promise executor receives `resolve` and `reject` functions in this order.

- Similarly to `resolve(data)`, you can reject with data `reject(data)` and the `data` will be made available to the `.catch()` callback.

Practice / Classroom coding: Two mini-projects

MiniProject01 : The goal of this mini-project is to allow the user to search the spacecraft provided in the `data.js` file by typing the name of the spacecraft in the search input.

MiniProject02: In this mini-project, you will build a page that uses the GitHub API to list the repositories of a GitHub user.