8 - Promises, async/await

Callbacks

Many functions are provided by JavaScript host environments that allow you to schedule asynchronous actions. **In other words, actions that we initiate now, but they finish later.** For instance, one such function is the setTimeout function. There are other real-world examples of asynchronous actions, e.g. **loading scripts and modules** (we'll cover them in later chapters).
An example with the `loadScript()` function that loads a script with the given src:

```
function loadScript(src) {
  // creates a <script> tag and append it to the page
  // this causes the script with given src to start loading and run when complete
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}
```

It inserts into the document a new, dynamically created, tag `\<script src="…">` with the given src. The browser automatically starts loading it and executes when complete. We can use this function like this:

```
loadScript('/my/script.js');
```

The script is executed "asynchronously", as it starts loading now, but runs later, when the function has already finished. If there's any code below loadScript(…), it doesn't wait until the script loading finishes.
Let's say we need to use the new script as soon as it loads. It declares new functions, and we want to run them. But if we do that immediately after the loadScript(…) call, that wouldn't work:

```
loadScript('/my/script.js'); // the script has "function newFunction() {…}"

newFunction(); // no such function!
```

Naturally, the browser probably didn't have time to load the script. As of now, the loadScript function doesn't provide a way to track the load completion. The script loads and eventually runs, that's all. But we'd like to know when it happens, to use new functions and variables from that script. Let's add a callback function as a second argument to loadScript that should execute when the script loads:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
```

```
  script.onload = () => callback(script);

  document.head.append(script);
}
```

The onload event is described in the article Resource loading: onload and onerror, it basically executes a function after the script is loaded and executed. Now if we want to call new functions from the script, we should write that in the callback:

```
loadScript('/my/script.js', function() {
  // the callback runs after the script is loaded
  newFunction(); // so now it works
  ...
});
```

That's the idea: **the second argument is a function (usually anonymous) that runs when the action is completed.**

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
  script.onload = () => callback(script);
  document.head.append(script);
}

loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js
', script => {
  alert(`Cool, the script ${script.src} is loaded`);
  alert( _ ); // _ is a function declared in the loaded script
});
```

That's called a "callback-based" style of asynchronous programming. A function that does something asynchronously should provide a callback argument where we put the function to run after it's complete. Here we did it in `loadScript`, but of course it's a general approach.

Callback in callback

How can we load two scripts sequentially: the first one, and then the second one after it? The natural solution would be to put the second loadScript call inside the callback, like this:

```
loadScript('/my/script.js', function(script) {

  alert(`Cool, the ${script.src} is loaded, let's load one more`);

  loadScript('/my/script2.js', function(script) {
    alert(`Cool, the second script is loaded`);
  });

});
```

So, every new action is inside a callback. That's fine for few actions, but not good for many, so we'll see other variants soon.

Handling errors

In the above examples we didn't consider errors. What if the script loading fails? Our callback should be able to react on that. Here's an improved version of `loadScript` that tracks loading errors:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Script load error for
${src}`));

  document.head.append(script);
}
```

It calls `callback(null, script)` for successful load and `callback(error)` otherwise. Usage:

```
loadScript('/my/script.js', function(error, script) {
  if (error) {
    // handle error
  } else {
    // script loaded successfully
  }
});
```

Once again, the recipe that we used for loadScript is actually quite common. It's called the "error-first callback" style. The convention is:

- The first argument of the callback is reserved for an error if it occurs. Then callback(err) is called.
- The second argument (and the next ones if needed) are for the successful result. Then callback(null, result1, result2…) is called.

So the single callback function is used both for reporting errors and passing back results.

Pyramid of Doom

At first glance, it looks like a viable approach to asynchronous coding. And indeed it is. For one or maybe two nested calls it looks fine. But for multiple asynchronous actions that follow one after another, we'll have code like this:

```
loadScript('1.js', function(error, script) {

  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // ...continue after all scripts are loaded (*)
          }
        });

      }
    });
  }
});
```

As calls become more nested, the code becomes deeper and increasingly more difficult to manage, especially if we have real code instead of ... that may include more loops, conditional statements and so on. That's sometimes called "callback hell" or "pyramid of doom." The "pyramid" of nested calls grows to the right with every asynchronous action. Soon it spirals out of control. So this way of coding isn't very good. We can try to alleviate the problem by making every action a standalone function, like this:

```
loadScript('1.js', step1);

function step1(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
```

```
    loadScript('2.js', step2);
  }
}

function step2(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('3.js', step3);
  }
}

function step3(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...continue after all scripts are loaded (*)
  }
}
```

See? It does the same thing, and there's no deep nesting now because we made every action a separate top-level function. It works, but the code looks like a torn apart spreadsheet. It's difficult to read, and you probably noticed that one needs to eye-jump between pieces while reading it. That's inconvenient, especially if the reader is not familiar with the code and doesn't know where to eye-jump. Also, the functions named step* are all of single use, they are created only to avoid the "pyramid of doom." No one is going to reuse them outside of the action chain. So there's a bit of namespace cluttering here. We'd like to have something better. Luckily, there are other ways to avoid such pyramids. One of the best ways is to use "promises".

Promise

Imagine that you're a top singer, and fans ask day and night for your upcoming song.

To get some relief, you promise to send it to them when it's published. You give your fans a list. They can fill in their email addresses, so that when the song becomes available, all subscribed parties instantly receive it. And even if something goes very wrong, say, a fire in the studio, so that you can't publish the song, they will still be notified. Everyone is happy: you, because the people don't crowd you anymore, and fans, because they won't miss the song. This is a real-life analogy for things we often have in programming:

- A "producing code" that does something and takes time. For instance, some code that loads the data over a network. That's a "singer".
- A "consuming code" that wants the result of the "producing code" once it's ready. Many functions may need that result. These are the "fans".
- A promise is a special JavaScript object that links the "producing code" and the "consuming code" together. In terms of our analogy: this is the "subscription list". The "producing code" takes whatever time it needs to produce the promised result, and the "promise" makes that result available to all of the subscribed code when it's ready.

The analogy isn't terribly accurate, because JavaScript promises are more complex than a simple subscription list: they have additional features and limitations. But it's fine to begin with. The constructor syntax for a promise object is:

```
let promise = new Promise(function(resolve, reject) {
  // executor (the producing code, "singer")
});
```

The function passed to new Promise is called the **executor**. When new Promise is created, the executor runs automatically. It contains the producing code which should eventually produce the result. In terms of the analogy above: the executor is the "singer". Its arguments resolve and reject are callbacks provided by JavaScript itself. Our code is only inside the executor.

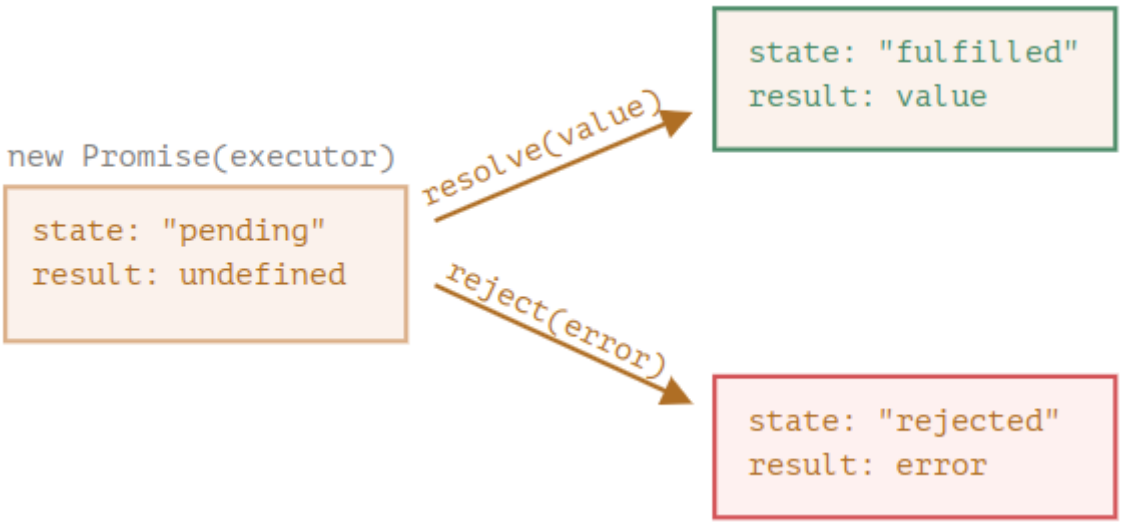
When the executor obtains the result, be it soon or late, doesn't matter, it should call one of these callbacks:

- resolve(value) — if the job is finished successfully, with result value.
- reject(error) — if an error has occurred, error is the error object.

So to summarize: the executor runs automatically and attempts to perform a job. When it is finished with the attempt, **it calls resolve if it was successful or reject if there was an error.**
The promise object returned by the new Promise constructor has these internal properties:

- state — initially "pending", then changes to either "fulfilled" when resolve is called or "rejected" when reject is called.

- result — initially undefined, then changes to value when resolve(value) is called or error when reject(error) is called.




```
let promise = new Promise(function(resolve, reject) {
  // the function is executed automatically when the promise is constructed

  // after 1 second signal that the job is done with the result "done"
  setTimeout(() => resolve("done"), 1000);
});
```

We can see two things by running the code above:

- The executor is called automatically and immediately (by new Promise).
- The executor receives two arguments: resolve and reject. These functions are pre-defined by the JavaScript engine, so we don't need to create them. We should only call one of them when ready.

After one second of "processing", the executor calls resolve("done") to produce the result. This changes the state of the promise object. That was an example of a successful job completion, a "fulfilled promise". And now an example of the executor rejecting the promise with an error:

```
let promise = new Promise(function(resolve, reject) {
  // after 1 second signal that the job is finished with an error
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});
```

To summarize, the executor should perform a job (usually something that takes time) and then call resolve or reject to change the state of the corresponding promise object. A promise that is either resolved or rejected is called **"settled"**, as opposed to an initially "pending" promise.

There can be only a single result or an error: The executor should call only one resolve or one reject. Any state change is final. All further calls of resolve and reject are ignored:

```
let promise = new Promise(function(resolve, reject) {
  resolve("done");

  reject(new Error("…")); // ignored
  setTimeout(() => resolve("…")); // ignored
});
```

The idea is that a job done by the executor may have only one result or an error. Also, resolve/reject expect only one argument (or none) and will ignore additional arguments.

Immediately calling resolve/reject: In practice, an executor usually does something asynchronously and calls resolve/reject after some time, but it doesn't have to. We also can call resolve or reject immediately, like this:

```
let promise = new Promise(function(resolve, reject) {
  // not taking our time to do the job
  resolve(123); // immediately give the result: 123
});
```

For instance, this might happen when we start to do a job but then see that everything has already been completed and cached. That's fine. We immediately have a resolved promise.

The state and result are internal: The properties state and result of the Promise object are internal. We can't directly access them. We can use the methods .then/.catch/.finally for that. They are described below.

Consumers: then, catch

A Promise object serves as a link between the executor (the “producing code” or “singer”) and the consuming functions (the “fans”), which will receive the result or error. Consuming functions can be registered (subscribed) using the methods .then and .catch.

then: the most important is .then

```
promise.then(
  function(result) { /* handle a successful result */ },
  function(error) { /* handle an error */ }
);
```

The first argument of **.then** is a function that runs when the promise is resolved and receives the result. The second argument of .then is a function that runs when the promise is rejected and receives the error. For instance, here’s a reaction to a successfully resolved promise:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done!"), 1000);
});

// resolve runs the first function in .then
promise.then(
  result => alert(result), // shows "done!" after 1 second
  error => alert(error) // doesn't run
);
```

The first function is executed, and in case of a rejection the second one:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// reject runs the second function in .then
promise.then(
  result => alert(result), // doesn't run
  error => alert(error) // shows "Error: Whoops!" after 1 second
);
```

If we’re interested only in successful completions, then we can provide only one function argument to .then:

```
let promise = new Promise(resolve => {
  setTimeout(() => resolve("done!"), 1000);
});
```

```
promise.then(alert); // shows "done!" after 1 second
```

.catch: If we're interested only in errors, then we can use null as the first argument: .then(null, errorHandlingFunction). Or we can use .catch(errorHandlingFunction), which is exactly the same

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// .catch(f) is the same as promise.then(null, f)
promise.catch(alert); // shows "Error: Whoops!" after 1 second
```

The call .catch(f) is a complete analog of .then(null, f), it's just a shorthand.

Cleanup: finally

Just like there's a finally clause in a regular try {...} catch {...}, there's finally in promises. The call .finally(f) is similar to .then(f, f) in the sense that f runs always, when the promise is settled: be it resolve or reject. The idea of finally is to set up a handler for performing cleanup/finalizing after the previous operations are complete. E.g. stopping loading indicators, closing no longer needed connections, etc. Think of it as a party finisher. No matter was a party good or bad, how many friends were in it, we still need (or at least should) do a cleanup after it.
The code may look like this:

```
new Promise((resolve, reject) => {
  /* do something that takes time, and then call resolve or maybe reject */
})
  // runs when the promise is settled, doesn't matter successfully or not
  .finally(() => stop loading indicator)
  // so the loading indicator is always stopped before we go on
  .then(result => show result, err => show error)
```

Please note that finally(f) isn't exactly an alias of then(f,f) though.

There are important differences:

- A finally handler has no arguments. In finally we don't know whether the promise is successful or not. That's all right, as our task is usually to perform "general" finalizing procedures. lease take a look at the example above: as you can see, the finally handler has no arguments, and the promise outcome is handled by the next handler.
- A finally handler "passes through" the result or error to the next suitable handler.

  For instance, here the result is passed through finally to then:

  ```
  new Promise((resolve, reject) => {
    setTimeout(() => resolve("value"), 2000);
  })
    .finally(() => alert("Promise ready")) // triggers first
    .then(result => alert(result)); // <-- .then shows "value"

  // An example with an error
  new Promise((resolve, reject) => {
    throw new Error("error");
  })
    .finally(() => alert("Promise ready")) // triggers first
    .catch(err => alert(err));  // <-- .catch shows the error
  ```

  As you can see, the value returned by the first promise is passed through finally to the next then.

That's very convenient, because finally is not meant to process a promise result. As said, it's a place to do generic cleanup, no matter what the outcome was.

- A finally handler also shouldn't return anything. If it does, the returned value is silently ignored. The only exception to this rule is when a finally handler throws an error. Then this error goes to the next handler, instead of any previous outcome.

To summarize:

- A finally handler doesn't get the outcome of the previous handler (it has no arguments). This outcome is passed through instead, to the next suitable handler.
- If a finally handler returns something, it's ignored.
- When finally throws an error, then the execution goes to the nearest error handler.

These features are helpful and make things work just the right way if we use finally how it's supposed to be used: for generic cleanup procedures.

We can attach handlers to settled promises: If a promise is pending, .then/catch/finally handlers wait for its outcome. Sometimes, it might be that a promise is already settled when we add a handler to it. In such case, these handlers just run immediately:

```
// the promise becomes resolved immediately upon creation
let promise = new Promise(resolve => resolve("done!"));
promise.then(alert); // done! (shows up right now)
```

Note that this makes promises more powerful than the real life "subscription list" scenario. If the singer has already released their song and then a person signs up on the subscription list, they probably won't receive that song. Subscriptions in real life must be done prior to the event. Promises are more flexible. We can add handlers any time: if the result is already there, they just execute.

Example: load script

Next, let's see more practical examples of how promises can help us write asynchronous code. We've got the loadScript function for loading a script from the previous chapter. Here's the callback-based variant, just to remind us of it:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Script load error for
${src}`));

  document.head.append(script);
}
```

Let's rewrite it using Promises. The new function loadScript will not require a callback. Instead, it will create and return a Promise object that resolves when the loading is complete. The outer code can add handlers (subscribing functions) to it using .then:

```
function loadScript(src) {
  return new Promise(function(resolve, reject) {
    let script = document.createElement('script');
    script.src = src;

    script.onload = () => resolve(script);
    script.onerror = () => reject(new Error(`Script load error for
${src}`));

    document.head.append(script);
  });
}
```

```
let promise =
loadScript("https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.
js");
promise.then(
  script => alert(`${script.src} is loaded!`),
  error => alert(`Error: ${error.message}`)
);

promise.then(script => alert('Another handler...'));
```

We can immediately see benefits over the callback based pattern:

- Promises allow us to do things in the natural order. First, we run loadScript(script), and .then we write what to do with the result. In Callbacks we must have a callback function at our disposal when calling loadScript(script, callback). In other words, we must know what to do with the result before loadScript is called.
- In promises we can call .then on a Promise as many times as we want. Each time, we're adding a new "fan", a new subscribing function, to the "subscription list". More about this in the next chapter: Promises chaining. Using callbacks there can be only one callback.

So promises give us better code flow and flexibility.

Exercises for this section can be found at: https://javascript.info/promise-basics#tasks

Promises chaining

Let's return to the problem mentioned in the chapter Introduction: callbacks: we have a sequence of asynchronous tasks to be performed one after another — for instance, loading scripts. How can we code it well? Promises provide a couple of recipes to do that. In this chapter we cover promise chaining. It looks like this:

```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000); // (*)

}).then(function(result) { // (**)

  alert(result); // 1
  return result * 2;

}).then(function(result) { // (***)

  alert(result); // 2
  return result * 2;

}).then(function(result) {

  alert(result); // 4
  return result * 2;

});
```

The idea is that the result is passed through the chain of .then handlers.

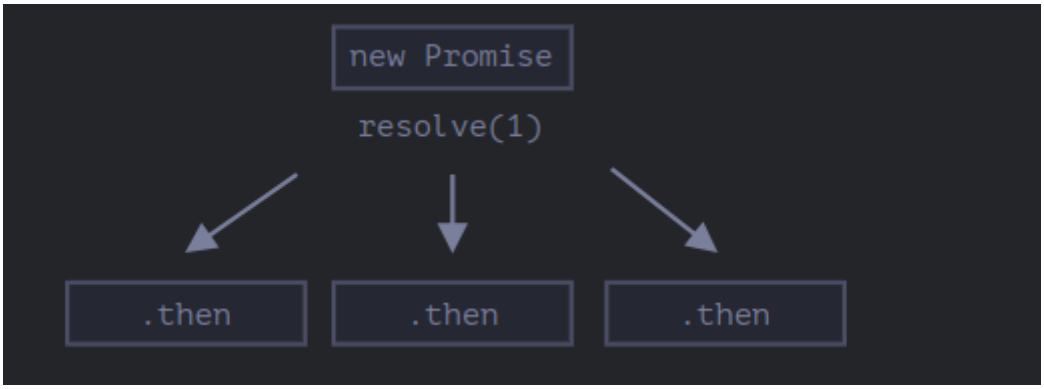
Here the flow is:

- The initial promise resolves in 1 second (*),
- Then the .then handler is called (**), which in turn creates a new promise (resolved with 2 value).
- The next then (***) gets the result of the previous one, processes it (doubles) and passes it to the next handler…and so on.
- As the result is passed along the chain of handlers, we can see a sequence of alert calls: 1 → 2 → 4. The whole thing works, because every call to a .then returns a new promise, so that we can call the next .then on it. When a handler returns a value, it becomes the result of that promise, so the next .then is called with it. **A classic newbie error: technically we can also add many .then to a single promise. This is not chaining.**

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
```

```
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});
```

What we did here is just several handlers to one promise. They don't pass the result to each other; instead they process it independently. Here's the picture (compare it with the chaining above):



All .then on the same promise get the same result – the result of that promise. So in the code above all alert show the same: 1. In practice we rarely need multiple handlers for one promise. Chaining is used much more often.

Returning promises

A handler, used in .then(handler) may create and return a promise. In that case further handlers wait until it settles, and then get its result.
For instance:

```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000);

}).then(function(result) {

  alert(result); // 1

  return new Promise((resolve, reject) => { // (*)
    setTimeout(() => resolve(result * 2), 1000);
  });

}).then(function(result) { // (**)

  alert(result); // 2

  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(result * 2), 1000);
  });

}).then(function(result) {

  alert(result); // 4

});
```

Here the first .then shows 1 and returns new Promise(…) in the line (*). After one second it resolves, and the result (the argument of resolve, here it's result * 2) is passed on to the handler of the second .then. That handler is in the line (**), it shows 2 and does the same thing. So the output is the same as in the previous example: 1 → 2 → 4, but now with 1 second delay between alert calls. Returning promises allows us to build chains of asynchronous actions.

Example: loadScript

```
loadScript("/article/promise-chaining/one.js")
  .then(function(script) {
    return loadScript("/article/promise-chaining/two.js");
  })
```

```
  .then(function(script) {
    return loadScript("/article/promise-chaining/three.js");
  })
  .then(function(script) {
    // use functions declared in scripts
    // to show that they indeed loaded
    one();
    two();
    three();
  });

    // a bit shorter with arrow functions

    loadScript("/article/promise-chaining/one.js")
  .then(script => loadScript("/article/promise-chaining/two.js"))
  .then(script => loadScript("/article/promise-chaining/three.js"))
  .then(script => {
    // scripts are loaded, we can use functions declared there
    one();
    two();
    three();
  });
```

Here each loadScript call returns a promise, and the next .then runs when it resolves. Then it initiates the loading of the next script. So scripts are loaded one after another. We can add more asynchronous actions to the chain. Please note that the code is still “flat” — it grows down, not to the right. There are no signs of the “pyramid of doom”.

Thenables

To be precise, a handler may return not exactly a promise, but a so-called “thenable” object – an arbitrary object that has a method .then. It will be treated the same way as a promise. The idea is that 3rd-party libraries may implement “promise-compatible” objects of their own. They can have an extended set of methods, but also be compatible with native promises, because they implement .then. Here’s an example of a thenable object:

```
class Thenable {
  constructor(num) {
    this.num = num;
  }
  then(resolve, reject) {
    alert(resolve); // function() { native code }
    // resolve with this.num*2 after the 1 second
    setTimeout(() => resolve(this.num * 2), 1000); // (**)
```

```
  }
}

new Promise(resolve => resolve(1))
  .then(result => {
    return new Thenable(result); // (*)
  })
  .then(alert); // shows 2 after 1000ms
```

JavaScript checks the object returned by the .then handler in line (*): if it has a callable method named then, then it calls that method providing native functions resolve, reject as arguments (similar to an executor) and waits until one of them is called. In the example above resolve(2) is called after 1 second (**). **Then the result is passed further down the chain. This feature allows us to integrate custom objects with promise chains without having to inherit from Promise.**

Bigger example: fetch

In frontend programming, promises are often used for network requests. So let's see an extended example of that. We'll use the fetch method to load the information about the user from the remote server. It has a lot of optional parameters covered in separate chapters, but the basic syntax is quite simple:

```
let promise = fetch(url);
```

This makes a network request to the url and returns a promise. The promise resolves with a response object when the remote server responds with headers, but before the full response is downloaded. To read the full response, we should call the method response.text(): it returns a promise that resolves when the full text is downloaded from the remote server, with that text as a result. The code below makes a request to user.json and loads its text from the server:

```
fetch('/article/promise-chaining/user.json')
  // .then below runs when the remote server responds
  .then(function(response) {
    // response.text() returns a new promise that resolves with the full response text
    // when it loads
    return response.text();
  })
  .then(function(text) {
    // ...and here's the content of the remote file
    alert(text); // {"name": "iliakan", "isAdmin": true}
  });
```

The response object returned from fetch also includes the method response.json() that reads the remote data and parses it as JSON. In our case that's even more convenient, so let's switch to it.

We'll also use arrow functions for brevity:

```
// same as above, but response.json() parses the remote content as JSON
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => alert(user.name)); // iliakan, got user name
```

Now let's do something with the loaded user. For instance, we can make one more request to GitHub, load the user profile and show the avatar:

```
// Make a request for user.json
fetch('/article/promise-chaining/user.json')
  // Load it as json
  .then(response => response.json())
  // Make a request to GitHub
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  // Load the response as json
  .then(response => response.json())
  // Show the avatar image (githubUser.avatar_url) for 3 seconds (maybe
animate it)
  .then(githubUser => {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => img.remove(), 3000); // (*)
  });
```

The code works; see comments about the details. However, there's a potential problem in it, a typical error for those who begin to use promises. Look at the line (*): how can we do something after the avatar has finished showing and gets removed? For instance, we'd like to show a form for editing that user or something else. As of now, there's no way. To make the chain extendable, we need to return a promise that resolves when the avatar finishes showing. Like this:

```
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  .then(response => response.json())
  .then(githubUser => new Promise(function(resolve, reject) { // (*)
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);
```

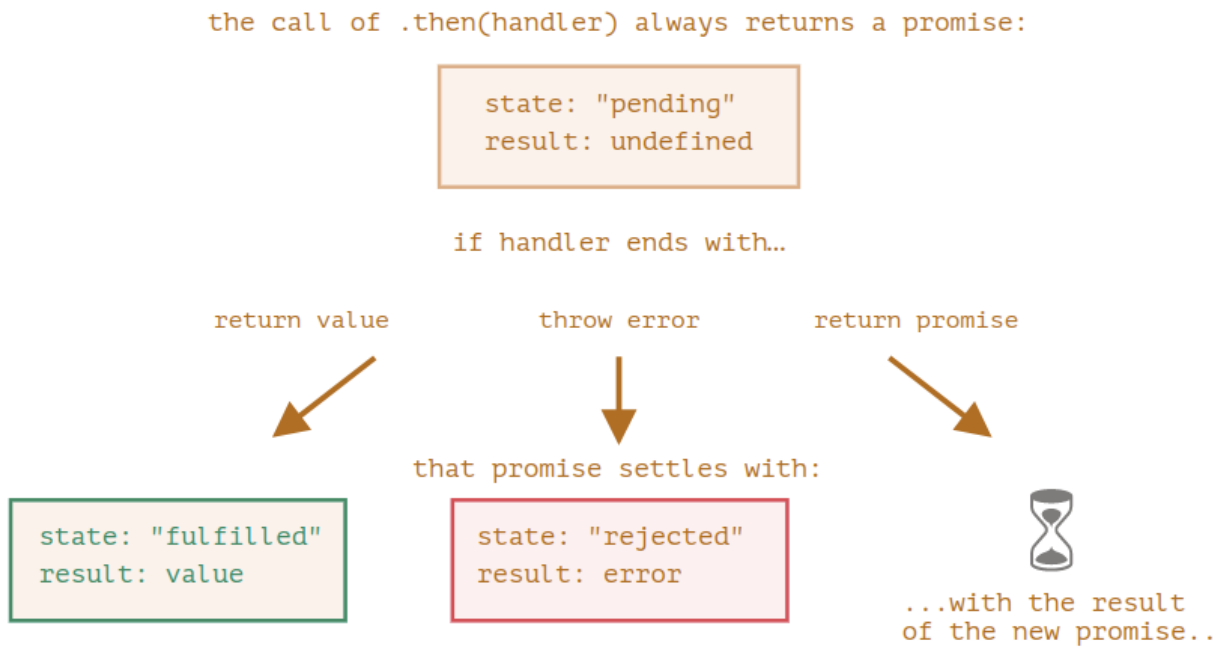
```
    setTimeout(() => {
      img.remove();
      resolve(githubUser); // (**)
    }, 3000);
  }))
  // triggers after 3 seconds
  .then(githubUser => alert(`Finished showing ${githubUser.name}`));
```

That is, the .then handler in line (*) now returns new Promise, that becomes settled only after the call of resolve(githubUser) in setTimeout (**). The next .then in the chain will wait for that. **As a good practice, an asynchronous action should always return a promise**. That makes it possible to plan actions after it; even if we don't plan to extend the chain now, we may need it later. Finally, we can split the code into reusable functions:

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

function loadGithubUser(name) {
  return loadJson(`https://api.github.com/users/${name}`);
}

function showAvatar(githubUser) {
  return new Promise(function(resolve, reject) {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => {
      img.remove();
      resolve(githubUser);
    }, 3000);
  });
}

// Use them:
loadJson('/article/promise-chaining/user.json')
  .then(user => loadGithubUser(user.name))
  .then(showAvatar)
  .then(githubUser => alert(`Finished showing ${githubUser.name}`));
  // ...
```

Summary

If a .then (or catch/finally, doesn't matter) handler returns a promise, the rest of the chain waits until it settles. When it does, its result (or error) is passed further. Here's a full picture:



Exercises for this section can be found at: https://javascript.info/promise-chaining#tasks

Error handling with promises

Promise chains are great at error handling. When a promise rejects, the control jumps to the closest rejection handler. That's very convenient in practice. For instance, in the code below the URL to fetch is wrong (no such site) and .catch handles the error:

```
fetch('https://no-such-server.blabla') // rejects
  .then(response => response.json())
  .catch(err => alert(err)) // TypeError: failed to fetch (the text may vary)
```

As you can see, the .catch doesn't have to be immediate. It may appear after one or maybe several .then. Or, maybe, everything is all right with the site, but the response is not valid JSON. The easiest way to catch all errors is to append .catch to the end of chain:

```
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  .then(response => response.json())
  .then(githubUser => new Promise((resolve, reject) => {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
```

```
    document.body.append(img);

    setTimeout(() => {
      img.remove();
      resolve(githubUser);
    }, 3000);
  }))
  .catch(error => alert(error.message));
```

Normally, such .catch doesn't trigger at all. But if any of the promises above rejects (a network problem or invalid json or whatever), then it would catch it.

Implicit try...catch

The code of a promise executor and promise handlers has an "invisible try..catch" around it. If an exception happens, it gets caught and treated as a rejection. For instance, this code:

```
new Promise((resolve, reject) => {
  throw new Error("Whoops!");
}).catch(alert); // Error: Whoops!

// Is the same as

new Promise((resolve, reject) => {
  reject(new Error("Whoops!"));
}).catch(alert); // Error: Whoops!
```

…Works exactly the same as this:

```
new Promise((resolve, reject) => {
  reject(new Error("Whoops!"));
}).catch(alert); // Error: Whoops!
```

The "invisible try..catch" around the executor automatically catches the error and turns it into rejected promise. This happens not only in the executor function, but in its handlers as well. If we throw inside a .then handler, that means a rejected promise, so the control jumps to the nearest error handler. Here's an example:

```
new Promise((resolve, reject) => {
  resolve("ok");
}).then((result) => {
  throw new Error("Whoops!"); // rejects the promise
}).catch(alert); // Error: Whoops!

// This happens for all errors, like programming ones:
```

```
new Promise((resolve, reject) => {
  resolve("ok");
}).then((result) => {
  blabla(); // no such function
}).catch(alert); // ReferenceError: blabla is not defined
```

The final .catch not only catches explicit rejections, but also accidental errors in the handlers above.

Rethrowing

As we already noticed, .catch at the end of the chain is similar to try..catch. We may have as many .then handlers as we want, and then use a single .catch at the end to handle errors in all of them. In a regular try..catch we can analyze the error and maybe rethrow it if it can't be handled. The same thing is possible for promises. If we throw inside .catch, then the control goes to the next closest error handler. And if we handle the error and finish normally, then it continues to the next closest successful .then handler. In the example below the .catch successfully handles the error:

```
// the execution: catch -> then
new Promise((resolve, reject) => {

  throw new Error("Whoops!");

}).catch(function(error) {

  alert("The error is handled, continue normally");

}).then(() => alert("Next successful handler runs"));
```

Here the .catch block finishes normally. So the next successful .then handler is called. In the example below we see the other situation with .catch. The handler (*) catches the error and just can't handle it (e.g. it only knows how to handle URIError), so it throws it again:

```
// the execution: catch -> catch
new Promise((resolve, reject) => {

  throw new Error("Whoops!");

}).catch(function(error) { // (*)

  if (error instanceof URIError) {
    // handle it
  } else {
    alert("Can't handle such error");

    throw error; // throwing this or another error jumps to the next catch
  }

}).then(function() {
  /* doesn't run here */
}).catch(error => { // (**)

  alert(`The unknown error has occurred: ${error}`);
```

```
  // don't return anything => execution goes the normal way

});
```

The execution jumps from the first .catch (*) to the next one (**) down the chain.

Unhandled rejections

What happens when an error is not handled? For instance, we forgot to append .catch to the end of the chain, like here:

```
new Promise(function() {
  noSuchFunction(); // Error here (no such function)
})
  .then(() => {
    // successful promise handlers, one or more
  }); // without .catch at the end!
```

In case of an error, the promise becomes rejected, and the execution should jump to the closest rejection handler. But there is none. So the error gets “stuck”. There’s no code to handle it. In practice, just like with regular unhandled errors in code, it means that something has gone terribly wrong. What happens when a regular error occurs and is not caught by try..catch? The script dies with a message in the console. A similar thing happens with unhandled promise rejections. The JavaScript engine tracks such rejections and generates a global error in that case. You can see it in the console if you run the example above. In the browser we can catch such errors using the event unhandledrejection:

```
window.addEventListener('unhandledrejection', function(event) {
  // the event object has two special properties:
  alert(event.promise); // [object Promise] - the promise that generated the
error
  alert(event.reason); // Error: Whoops! - the unhandled error object
});

new Promise(function() {
  throw new Error("Whoops!");
}); // no catch to handle the error
```

The event is the part of the HTML standard. If an error occurs, and there’s no .catch, the unhandledrejection handler triggers, and gets the event object with the information about the error, so we can do something. Usually such errors are unrecoverable, so our best way out is to inform the user about the problem and probably report the incident to the server. In non-browser environments like Node.js there are other ways to track unhandled errors.

Summary

- .catch handles errors in promises of all kinds: be it a reject() call, or an error thrown in a handler.

- .then also catches errors in the same manner, if given the second argument (which is the error handler).
- We should place .catch exactly in places where we want to handle errors and know how to handle them. The handler should analyze errors (custom error classes help) and rethrow unknown ones (maybe they are programming mistakes).
  It's ok not to use .catch at all, if there's no way to recover from an error.
  In any case we should have the unhandledrejection event handler (for browsers, and analogs for other environments) to track unhandled errors and inform the user (and probably our server) about them, so that our app never "just dies".

Exercises for this section can be found at: https://javascript.info/promise-error-handling#error-in-settimeout

Promise API

There are 6 static methos in the **Promise** class:

Promise.all

Let's say we want many promises to execute in parallel and wait until all of them are ready. For instance, download several URLs in parallel and process the content once they are all done. That's what Promise.all is for. The syntax is:

```
let promise = Promise.all(iterable);
```

Promise.all takes an iterable (usually, an array of promises) and returns a new promise. The new promise resolves when all listed promises are resolved, and the array of their results becomes its result. For instance, the Promise.all below settles after 3 seconds, and then its result is an array [1, 2, 3]:

```
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
  new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2
  new Promise(resolve => setTimeout(() => resolve(3), 1000))  // 3
]).then(alert); // 1,2,3 when promises are ready: each promise contributes an array member
```

Please note that the order of the resulting array members is the same as in its source promises. Even though the first promise takes the longest time to resolve, it's still first in the array of results. A common trick is to map an array of job data into an array of promises, and then wrap that into Promise.all. For instance, if we have an array of URLs, we can fetch them all like this:

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://api.github.com/users/jeresig'
];

// map every url to the promise of the fetch
let requests = urls.map(url => fetch(url));

// Promise.all waits until all jobs are resolved
Promise.all(requests)
  .then(responses => responses.forEach(
    response => alert(`${response.url}: ${response.status}`)
  ));
```

A bigger example with fetching user information for an array of GitHub users by their names (we could fetch an array of goods by their ids, the logic is identical):

```
let names = ['iliakan', 'remy', 'jeresig'];

let requests = names.map(name =>
fetch(`https://api.github.com/users/${name}`));

Promise.all(requests)
  .then(responses => {
    // all responses are resolved successfully
    for(let response of responses) {
      alert(`${response.url}: ${response.status}`); // shows 200 for every
url
    }

    return responses;
  })
  // map array of responses into an array of response.json() to read their
content
  .then(responses => Promise.all(responses.map(r => r.json())))
  // all JSON answers are parsed: "users" is the array of them
  .then(users => users.forEach(user => alert(user.name)));

// If any promise is rejected, the promise returned by Promise.all
immediately rejects with that error

Promise.all([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new
Error("Whoops!")), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).catch(alert); // Error: Whoops!
```

Here the second promise rejects in two seconds. That leads to an immediate rejection of Promise.all, so .catch executes: the rejection error becomes the outcome of the entire Promise.all.

In case of an error, other promises are ignored:

- If one promise rejects, Promise.all immediately rejects, completely forgetting about the other ones in the list. Their results are ignored.
- For example, if there are multiple fetch calls, like in the example above, and one fails, the others will still continue to execute, but Promise.all won't watch them anymore. They will probably settle, but their results will be ignored.

- Promise.all does nothing to cancel them, as there's no concept of "cancellation" in promises. In another chapter we'll cover AbortController that can help with that, but it's not a part of the Promise API.

Promise.all(iterable) allows non-promise "regular" values in iterable

Normally, `Promise.all(...)` accepts an iterable (in most cases an array) of promises. But if any of those objects is not a promise, it's passed to the resulting array "as is". For instance, here the results are [1, 2, 3]:

```
Promise.all([
  new Promise((resolve, reject) => {
    setTimeout(() => resolve(1), 1000)
  }),
  2,
  3
]).then(alert); // 1, 2, 3
```

So we are able to pass ready values to Promise.all where convenient.

Promise.allSettled (recent addition)

Promise.all rejects as a whole if any promise rejects. That's good for "all or nothing" cases, when we need all results successful to proceed:

```
Promise.all([
  fetch('/template.html'),
  fetch('/style.css'),
  fetch('/data.json')
]).then(render); // render method needs results of all fetches
```

Promise.allSettled just waits for all promises to settle, regardless of the result. The resulting array has:

- {status:"fulfilled", value:result} for successful responses,
- {status:"rejected", reason:error} for errors.

For example, we'd like to fetch the information about multiple users. Even if one request fails, we're still interested in the others. Let's use **Promise.allSettled**:

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://no-such-url'
];

Promise.allSettled(urls.map(url => fetch(url)))
  .then(results => { // (*)
```

```
    results.forEach((result, num) => {
      if (result.status == "fulfilled") {
        alert(`${urls[num]}: ${result.value.status}`);
      }
      if (result.status == "rejected") {
        alert(`${urls[num]}: ${result.reason}`);
      }
    });
  });

// The array returned by the map function will be:

[
  {status: 'fulfilled', value: ...response...},
  {status: 'fulfilled', value: ...response...},
  {status: 'rejected', reason: ...error object...}
]
```

So for each promise we get its status and value/error.

Polyfill

If the browser doesn't support Promise.allSettled, it's easy to polyfill:

```
if (!Promise.allSettled) {
  const rejectHandler = reason => ({ status: 'rejected', reason });

  const resolveHandler = value => ({ status: 'fulfilled', value });

  Promise.allSettled = function (promises) {
    const convertedPromises = promises.map(p =>
Promise.resolve(p).then(resolveHandler, rejectHandler));
    return Promise.all(convertedPromises);
  };
}
```

n this code, `promises.map` takes input values, turns them into promises (just in case a non-promise was passed) with p => `Promise.resolve(p)`, and then adds `.then` handler to every one. That handler turns a successful result value into `{status:'fulfilled', value}`, and an error reason into `{status:'rejected', reason}`. That's exactly the format of `Promise.allSettled`. Now we can use `Promise.allSettled` to get the results of all given promises, even if some of them reject.

Promise.race

Similar to Promise.all, but waits only for the first settled promise and gets its result (or error). The syntax is:

```
let promise = Promise.race(iterable);

// For instance, here the result will be 1:
Promise.race([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new
Error("Whoops!")), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).then(alert); // 1
```

The first promise here was fastest, so it became the result. After the first settled promise “wins the race”, all further results/errors are ignored.

Promise.any

Similar to Promise.race, but waits only for the first fulfilled promise and gets its result. If all of the given promises are rejected, then the returned promise is rejected with AggregateError – a special error object that stores all promise errors in its errors property.

```
let promise = Promise.any(iterable);

// Here the result will be 1 again:
Promise.any([
  new Promise((resolve, reject) => setTimeout(() => reject(new
Error("Whoops!")), 1000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).then(alert); // 1
```

The first promise here was fastest, but it was rejected, so the second promise became the result. After the first fulfilled promise “wins the race”, all further results are ignored. Here’s an example when all promises fail:

```
Promise.any([
  new Promise((resolve, reject) => setTimeout(() => reject(new
Error("Ouch!")), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new
Error("Error!")), 2000))
]).catch(error => {
  console.log(error.constructor.name); // AggregateError
  console.log(error.errors[0]); // Error: Ouch!
  console.log(error.errors[1]); // Error: Error!
});
```

As you can see, error objects for failed promises are available in the errors property of the AggregateError object.

Promise.resolve

Methods `Promise.resolve` and `Promise.reject` are rarely needed in modern code, because `async/await` syntax (we'll cover it a bit later) makes them somewhat obsolete. We cover them here for completeness and for those who can't use `async/await` for some reason.

Promise.resolve

creates a resolved promise with the result `value`, same as:

```
let promise = new Promise(resolve => resolve(value));
```

The method is used for compatibility, when a function is expected to return a promise. For example, the `loadCached` function below fetches a URL and remembers (caches) its content. For future calls with the same URL it immediately gets the previous content from cache, but uses `Promise.resolve` to make a promise of it, so the returned value is always a promise:

```
let cache = new Map();

function loadCached(url) {
  if (cache.has(url)) {
    return Promise.resolve(cache.get(url)); // (*)
  }

  return fetch(url)
    .then(response => response.text())
    .then(text => {
      cache.set(url,text);
      return text;
    });
}
```

We can write `loadCached(url).then(…)`, because the function is guaranteed to return a promise. We can always use `.then` after `loadCached`. That's the purpose of `Promise.resolve` in the line (*).

Promise.reject

Promise.reject(error) creates a rejected promise with error. Same as:

```
let promise = new Promise((resolve, reject) => reject(error));
```

In practice this method is almost never used.

Summary

There are 6 static methods of Promise class:

- `Promise.all(promises)` – waits for all promises to resolve and returns an array of their results. If any of the given promises rejects, it becomes the error of `Promise.all`, and all other results are ignored.
- `Promise.allSettled(promises)` (recently added method) – waits for all promises to settle and returns their results as an array of objects with:
  `status: "fulfilled"` or `"rejected"`
  `value` (if fulfilled) or `reason` (if rejected).
- `Promise.race(promises)` – waits for the first promise to settle, and its result/error becomes the outcome.
- `Promise.any(promises)` (recently added method) – waits for the first promise to fulfill, and its result becomes the outcome. If all of the given promises are rejected, `AggregateError` becomes the error of `Promise.any`.
- `Promise.resolve(value)` – makes a resolved promise with the given value.
- `Promise.reject(error)` – makes a rejected promise with the given error.
  Of all these, Promise.all is probably the most common in practice.

Promisification

“Promisification” is a long word for a simple transformation. It’s the conversion of a function that accepts a callback into a function that returns a promise. Such transformations are often required in real-life, as many functions and libraries are callback-based. But promises are more convenient, so it makes sense to promisify them. For better understanding, let’s see an example. For instance, we have **loadScript(src, callback)** from the chapter Introduction: callbacks.

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Script load error for
${src}`));

  document.head.append(script);
}

// usage:
// loadScript('path/script.js', (err, script) => {...})
```

The function loads a script with the given src, and then calls `callback(err)` in case of an error, or `callback(null, script)` in case of successful loading. That’s a widespread agreement for using callbacks, we saw it before. Let’s promisify it. We’ll make a new function `loadScriptPromise(src)`, that does the same (loads the script), but returns a promise instead of using callbacks. In other words, we pass it only `src` (no `callback`) and get a promise in return, that resolves with `script` when the load is successful, and rejects with the error otherwise.

```
let loadScriptPromise = function(src) {
  return new Promise((resolve, reject) => {
    loadScript(src, (err, script) => {
      if (err) reject(err);
      else resolve(script);
    });
  });
};

// usage:
// loadScriptPromise('path/script.js').then(...)
```

As we can see, the new function is a wrapper around the original loadScript function. It calls it providing its own callback that translates to promise resolve/reject. Now loadScriptPromise fits well in promise-based code. If we like promises more than callbacks (and soon we’ll see more reasons for that), then

we will use it instead. In practice we may need to promisify more than one function, so it makes sense to use a helper. We'll call it promisify(f): it accepts a to-promisify function f and returns a wrapper function.

```
function promisify(f) {
  return function (...args) { // return a wrapper-function (*)
    return new Promise((resolve, reject) => {
      function callback(err, result) { // our custom callback for f (**)
        if (err) {
          reject(err);
        } else {
          resolve(result);
        }
      }

      args.push(callback); // append our custom callback to the end of f
arguments

      f.call(this, ...args); // call the original function
    });
  };
}

// usage:
let loadScriptPromise = promisify(loadScript);
loadScriptPromise(...).then(...);
```

The code may look a bit complex, but it's essentially the same that we wrote above, while promisifying `loadScript` function .A call to `promisify(f)` returns a wrapper around f (*). That wrapper returns a promise and forwards the call to the original f, tracking the result in the custom callback (**). Here, promisify assumes that the original function expects a callback with exactly two arguments `(err, result)`. That's what we encounter most often. Then our custom callback is in exactly the right format, and promisify works great for such a case. But what if the original f expects a callback with more arguments `callback(err, res1, res2, ...)`? We can improve our helper. Let's make a more advanced version of `promisify`.

- When called as promisify(f) it should work similar to the version above.
- When called as promisify(f, true), it should return the promise that resolves with the array of callback results. That's exactly for callbacks with many arguments.

```
// promisify(f, true) to get array of results
function promisify(f, manyArgs = false) {
  return function (...args) {
    return new Promise((resolve, reject) => {
```

```
      function callback(err, ...results) { // our custom callback for f
        if (err) {
          reject(err);
        } else {
          // resolve with all callback results if manyArgs is specified
          resolve(manyArgs ? results : results[0]);
        }
      }

      args.push(callback);

      f.call(this, ...args);
    });
  };
}

// usage:
f = promisify(f, true);
f(...).then(arrayOfResults => ..., err => ...);
```

As you can see it's essentially the same as above, but `resolve` is called with only one or all arguments depending on whether `manyArgs` is truthy. For more exotic callback formats, like those without `err` at all: `callback(result)`, we can promisify such functions manually without using the helper. There are also modules with a bit more flexible promisification functions, e.g. es6-promisify. In Node.js, there's a built-in `util.promisify` function for that.

Please note:
Promisification is a great approach, especially when you use async/await (covered later in the chapter Async/await), but not a total replacement for callbacks. Remember, a promise may have only one result, but a callback may technically be called many times. So promisification is only meant for functions that call the callback once. Further calls will be ignored.