

→ Promises, `async/await`

# Promise

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

To get a relief, you promise to send it to them when it's published. You give your fans a list. They can fill in their coordinates, so that when the song becomes available, all subscribed parties instantly get it. And if something goes very wrong, so that the song won't be published ever, then they are also to be notified.

Everyone is happy: you, because the people don't crowd you any more, and fans, because they won't miss the single.

That was a real-life analogy for things we often have in programming:

1. A "producing code" that does something and needs time. For instance, it loads a remote script. That's a "singer".
2. A "consuming code" wants the result when it's ready. Many functions may need that result. These are "fans".
3. A *promise* is a special JavaScript object that links them together. That's a "list". The producing code creates it and gives to everyone, so that they can subscribe for the result.

The analogy isn't very accurate, because JavaScript promises are more complex than a simple list: they have additional features and limitations. But still they are alike.

The constructor syntax for a promise object is:

```
1 let promise = new Promise(function(resolve, reject) {  
2   // executor (the producing code, "singer")  
3 });
```

The function passed to `new Promise` is called *executor*. When the promise is created, it's called automatically. It contains the producing code, that should eventually finish with a result. In terms of the analogy above, the executor is a "singer".

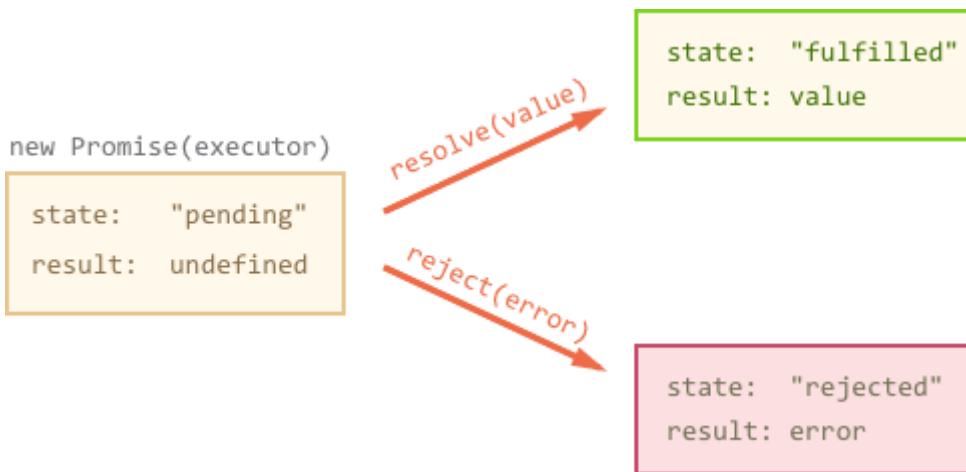
The resulting `promise` object has internal properties:

- `state` – initially is "pending", then changes to "fulfilled" or "rejected",
- `result` – an arbitrary value, initially `undefined`.

When the executor finishes the job, it should call one of:

- `resolve(value)` – to indicate that the job finished successfully:
  - sets `state` to "fulfilled",
  - sets `result` to `value`.
- `reject(error)` – to indicate that an error occurred:
  - sets `state` to "rejected",

- sets result to error.



Here's a simple executor, to gather that all together:

```

1 let promise = new Promise(function(resolve, reject) {
2   // the function is executed automatically when the promise is constructed
3
4   alert(resolve); // function () { [native code] }
5   alert(reject); // function () { [native code] }
6
7   // after 1 second signal that the job is done with the result "done!"
8   setTimeout(() => resolve("done!"), 1000);
9 });
  
```



We can see two things by running the code above:

1. The executor is called automatically and immediately (by `new Promise` ).
2. The executor receives two arguments: `resolve` and `reject` – these functions come from JavaScript engine. We don't need to create them. Instead the executor should call them when ready.

After one second of thinking the executor calls `resolve("done")` to produce the result:



That was an example of the “successful job completion”.

And now an example where the executor rejects promise with an error:

```

1 let promise = new Promise(function(resolve, reject) {
2   // after 1 second signal that the job is finished with an error
3   setTimeout(() => reject(new Error("Whoops!")), 1000);
4 });
  
```



result: undefined

result: error

To summarize, the executor should do a job (something that takes time usually) and then call `resolve` or `reject` to change the state of the corresponding promise object.

The promise that is either resolved or rejected is called "settled", as opposed to a "pending" promise.

### i There can be only one result or an error

The executor should call only one `resolve` or `reject`. The promise state change is final.

All further calls of `resolve` and `reject` are ignored:

```
1 let promise = new Promise(function(resolve, reject) {  
2   resolve("done");  
3  
4   reject(new Error("...")); // ignored  
5   setTimeout(() => resolve("...")); // ignored  
6 });
```

The idea is that a job done by the executor may have only one result or an error. In programming, there exist other data structures that allow many "flowing" results, for instance streams and queues. They have their own advantages and disadvantages versus promises. They are not supported by JavaScript core and lack certain language features that promises provide, we don't cover them here to concentrate on promises.

Also if we call `resolve/reject` with more than one argument – only the first argument is used, the next ones are ignored.

### i Reject with Error objects

Technically we can call `reject` (just like `resolve`) with any type of argument. But it's recommended to use `Error` objects in `reject` (or inherit from them). The reasoning for that will become obvious soon.

### i Resolve/reject can be immediate

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

```
1 let promise = new Promise(function(resolve, reject) {  
2   resolve(123); // immediately give the result: 123  
3 });
```

For instance, it happens when we start to do a job and then see that everything has already been done. Technically that's fine: we have a resolved promise right now.

## The state and result are internal

Properties `state` and `result` of a promise object are internal. We can't directly access them from our code, but we can use methods `.then/catch` for that, they are described below.

## Consumers: “`.then`” and “`.catch`”

A promise object serves as a link between the producing code (executor) and the consuming functions – those that want to receive the result/error. Consuming functions can be registered using methods `promise.then` and `promise.catch`.

The syntax of `.then` is:

```
1 promise.then(  
2   function(result) { /* handle a successful result */ },  
3   function(error) { /* handle an error */ }  
4 );
```

The first function argument runs when the promise is resolved and gets the result, and the second one – when it's rejected and gets the error.

For instance:

```
1 let promise = new Promise(function(resolve, reject) {  
2   setTimeout(() => resolve("done!"), 1000);  
3 });  
4  
5 // resolve runs the first function in .then  
6 promise.then(  
7   result => alert(result), // shows "done!" after 1 second  
8   error => alert(error) // doesn't run  
9 );
```



In case of a rejection:

```
1 let promise = new Promise(function(resolve, reject) {  
2   setTimeout(() => reject(new Error("Whoops!")), 1000);  
3 });  
4  
5 // reject runs the second function in .then  
6 promise.then(  
7   result => alert(result), // doesn't run  
8   error => alert(error) // shows "Error: Whoops!" after 1 second  
9 );
```



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

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



```
3 });
4
5 promise.then(alert); // shows "done!" after 1 second
```

If we're interested only in errors, then we can use `.then(null, function)` or an "alias" to it:  
`.catch(function)`

```
1 let promise = new Promise((resolve, reject) => {
2   setTimeout(() => reject(new Error("Whoops!")), 1000);
3 });
4
5 // .catch(f) is the same as promise.then(null, f)
6 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.

### i On settled promises then runs immediately

If a promise is pending, `.then/catch` handlers wait for the result. Otherwise, if a promise has already settled, they execute immediately:

```
1 // an immediately resolved promise
2 let promise = new Promise(resolve => resolve("done!"));
3
4 promise.then(alert); // done! (shows up right now)
```

That's handy for jobs that may sometimes require time and sometimes finish immediately. The handler is guaranteed to run in both cases.

## i Handlers of .then/catch are always asynchronous

To be even more precise, when `.then/catch` handler should execute, it first gets into an internal queue. The JavaScript engine takes handlers from the queue and executes when the current code finishes, similar to `setTimeout(..., 0)`.

In other words, when `.then(handler)` is going to trigger, it does something like `setTimeout(handler, 0)` instead.

In the example below the promise is immediately resolved, so `.then(alert)` triggers right now: the `alert` call is queued and runs immediately after the code finishes.

```
1 // an immediately resolved promise
2 let promise = new Promise(resolve => resolve("done!"));
3
4 promise.then(alert); // done! (right after the current code finishes)
5
6 alert("code finished"); // this alert shows first
```



So the code after `.then` always executes before the handler (even in the case of a pre-resolved promise). Usually that's unimportant, in some scenarios may matter.

Now let's see more practical examples how promises can help us in writing asynchronous code.

## Example: loadScript

We've got the `loadScript` function for loading a script from the previous chapter.

Here's the callback-based variant, just to remind it:

```
1 function loadScript(src, callback) {
2   let script = document.createElement('script');
3   script.src = src;
4
5   script.onload = () => callback(null, script);
6   script.onerror = () => callback(new Error(`Script load error ` + src));
7
8   document.head.append(script);
9 }
```

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 settles when the loading is complete. The outer code can add handlers to it using `.then`:

```
1 function loadScript(src) {
2   return new Promise(function(resolve, reject) {
3     let script = document.createElement('script');
4     script.src = src;
5
6     script.onload = () => resolve(script);
```



```
7     script.onerror = () => reject(new Error("Script load error: " + src));
8
9     document.head.append(script);
10    });
11 }
```

Usage:

```
1 let promise = loadScript("https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.20.0/loda
2
3 promise.then(
4   script => alert(`${script.src} is loaded!`),
5   error => alert(`Error: ${error.message}`)
6 );
7
8 promise.then(script => alert('One more handler to do something else!'));
```

We can immediately see few benefits over the callback-based syntax:

## Callbacks

- We must have a ready `callback` function when calling `loadScript`. In other words, we must know what to do with the result *before* `loadScript` is called.
- There can be only one callback.

## Promises

- Promises allow us to code things in the natural order. First we run `loadScript`, and `.then` write what to do with the result.
- We can call `.then` on a promise as many times as we want, at any time later.

So promises already give us better code flow and flexibility. But there's more. We'll see that in the next chapters.

## Tasks

### Re-resolve a promise? ↗

What's the output of the code below?

```
1 let promise = new Promise(function(resolve, reject) {
2   resolve(1);
3
4   setTimeout(() => resolve(2), 1000);
```

```
5 });
6
7 promise.then(alert);
```

solution

## Delay with a promise ↗

The built-in function `setTimeout` uses callbacks. Create a promise-based alternative.

The function `delay(ms)` should return a promise. That promise should resolve after `ms` milliseconds, so that we can add `.then` to it, like this:

```
1 function delay(ms) {
2   // your code
3 }
4
5 delay(3000).then(() => alert('runs after 3 seconds'));
```

solution

## Animated circle with promise ↗

Rewrite the `showCircle` function in the solution of the task [Animated circle with callback](#) so that it returns a promise instead of accepting a callback.

The new usage:

```
1 showCircle(150, 150, 100).then(div => {
2   div.classList.add('message-ball');
3   div.append("Hello, world!");
4 });
```

Take the solution of the task [Animated circle with callback](#) as the base.

solution



Previous lesson

Next lesson



## Comments

- You're welcome to post additions, questions to the articles and answers to them.
  - To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox ([plnkr](#), [JSBin](#), [codepen](#)...)
  - If you can't understand something in the article – please elaborate.
- 

© 2007—2017 Ilya Kantor

[contact us](#)

[about the project](#)

[RU / EN](#)

powered by [node.js](#) & [open source](#)









