August 14, 2022

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:

1. A "producing code" that does something and takes time. For instance, some code that loads the data over a network. That's a "singer".
2. 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".
3. 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:

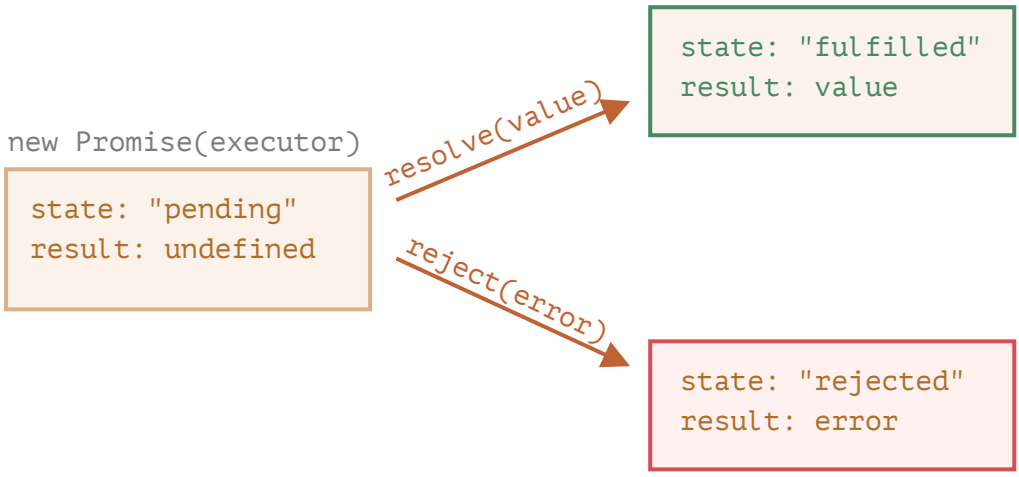
- `resolve(value)` — if the job is finished successfully, with result `value`.
- `reject(error)` — if an error has occurred, `error` is the error object.

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.

So the executor eventually moves `promise` to one of these states:



Later we'll see how "fans" can subscribe to these changes.

Here's an example of a promise constructor and a simple executor function with "producing code" that takes time (via `setTimeout`):

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

  // 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:

1. The executor is called automatically and immediately (by `new Promise`).
2. 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);
});
```

The call to `reject(...)` moves the promise object to `"rejected"` state:

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.

Reject with `Error` objects

In case something goes wrong, the executor should call `reject`. That can be done with any type of argument (just like `resolve`). But it is recommended to use `Error` objects (or objects that inherit from `Error`). The reasoning for that will soon become apparent.

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, fundamental one is `.then`.

The syntax is:

```
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 was executed.

And in the 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 no
  .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:

1. 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.

   Please 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.

2. 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"
```

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.

And here's an example of an error, for us to see how it's passed through `finally` to `catch`:

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

3. 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: loadScript

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);
  });
}
```

Usage:

```
let promise = loadScript("https://cdnjs.cloudflare.com/ajax/libs/lodash

promise.then(
  script => alert(`${script.src} is loaded!`),
  error => alert(`Error: ${error.message}`)
);

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

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

Promises	Callbacks
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.	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.
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.	There can be only one callback.

So promises 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?

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

  setTimeout(() => resolve(2), 1000);
});

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:

```
function delay(ms) {
  // your code
}

delay(3000).then(() => alert('runs after 3 seconds'));
```

solution