4th February 2020

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:

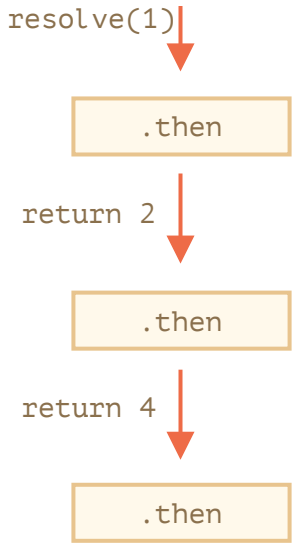
```
new Promise(function(resolve, reject) {

  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:

1. The initial promise resolves in 1 second `(*)`,
2. Then the `.then` handler is called `(**)`.
3. The value that it returns is passed to the next `.then` handler `(***)`
4. …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 a call to `promise.then` returns a 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.

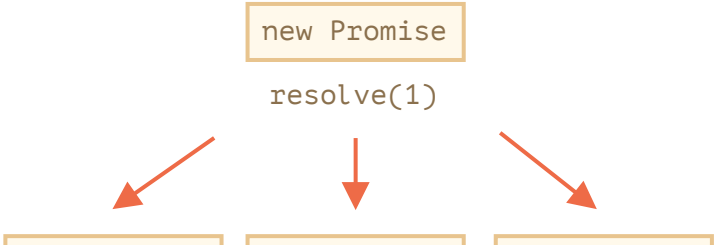
A classic newbie error: technically we can also add many `.then` to a single promise. This is not chaining.

For example:

```
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 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

Let's use this feature with the promisified `loadScript`, defined in the previous chapter, to load scripts one by one, in sequence:

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

This code can be made 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”.

Technically, we could add `.then` directly to each `loadScript`, like this:

```
loadScript("/article/promise-chaining/one.js").then(script1 => {
  loadScript("/article/promise-chaining/two.js").then(script2 => {
    loadScript("/article/promise-chaining/three.js").then(script3 => {
      // this function has access to variables script1, script2 and script3
      one();
      two();
      three();
    });
  });
});
```

This code does the same: loads 3 scripts in sequence. But it “grows to the right”. So we have the same problem as with callbacks.

People who start to use promises sometimes don’t know about chaining, so they write it this way. Generally, chaining is preferred.

Sometimes it’s ok to write `.then` directly, because the nested function has access to the outer scope. In the example above the most nested callback has access to all variables `script1`, `script2`, `script3`. But

that's an exception rather than a rule.

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:

```
1  class Thenable {
2    constructor(num) {
3      this.num = num;
4    }
5    then(resolve, reject) {
6      alert(resolve); // function() { native code }
7      // resolve with this.num*2 after the 1 second
8      setTimeout(() => resolve(this.num * 2), 1000); // (**)
9    }
10 }
11
12 new Promise(resolve => resolve(1))
13   .then(result => {
14     return new Thenable(result); // (*)
15   })
16   .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:

```
1  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 resp
    // 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 requests 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 anima
  .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:

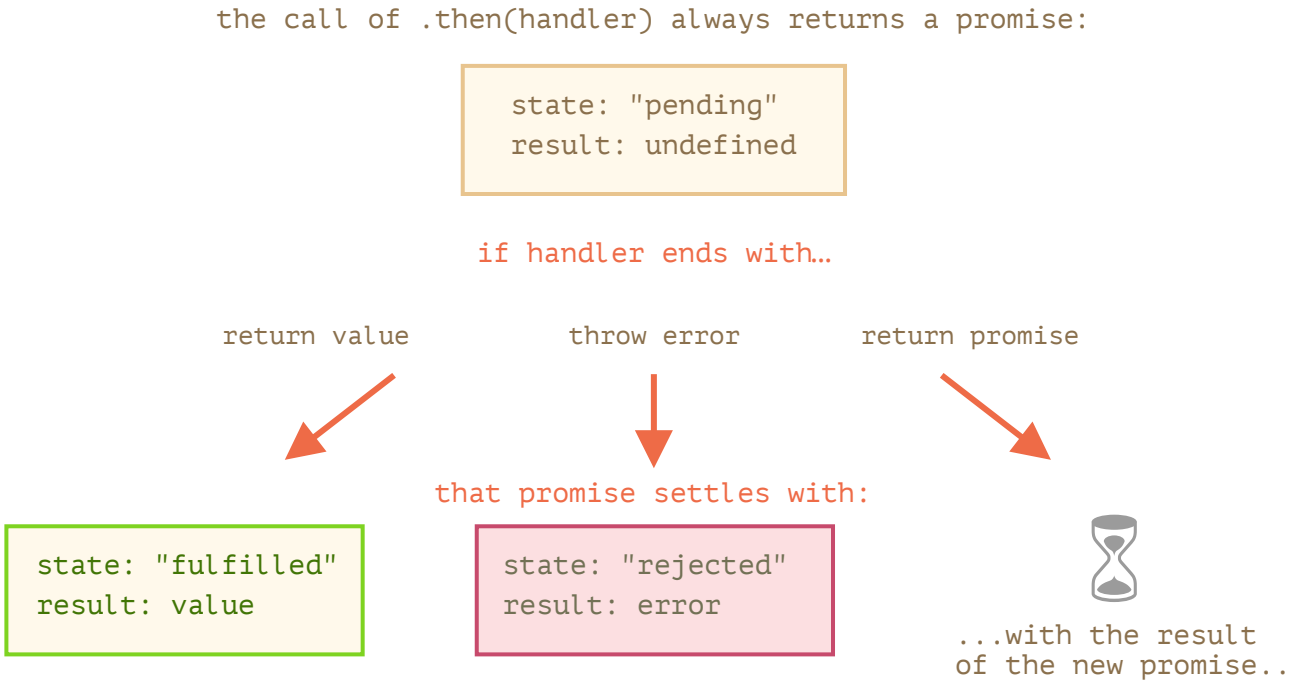
```
function loadJson(url) {
  return fetch(url)
    .then(response => response.json());
}

function loadGithubUser(name) {
  return fetch(`https://api.github.com/users/${name}`)
    .then(response => response.json());
}

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:



Tasks

Promise: then versus catch

Are these code fragments equal? In other words, do they behave the same way in any circumstances, for any handler functions?

```
promise.then(f1).catch(f2);
```

Versus:

```
promise.then(f1, f2);
```

solution

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- 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…)