Javascript

Promise Chaining and Fetch API

Promise Chaining

If 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 lecture we cover promise chaining.

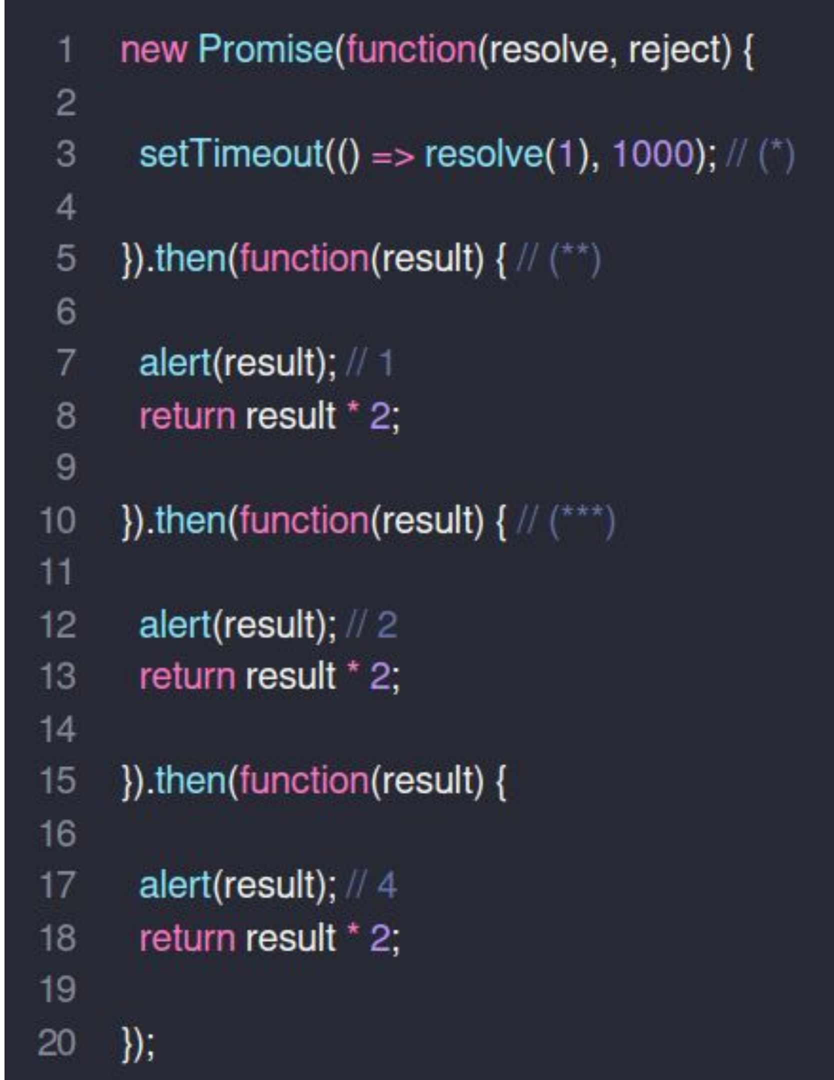
Simplified Promise Chaining Example

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.

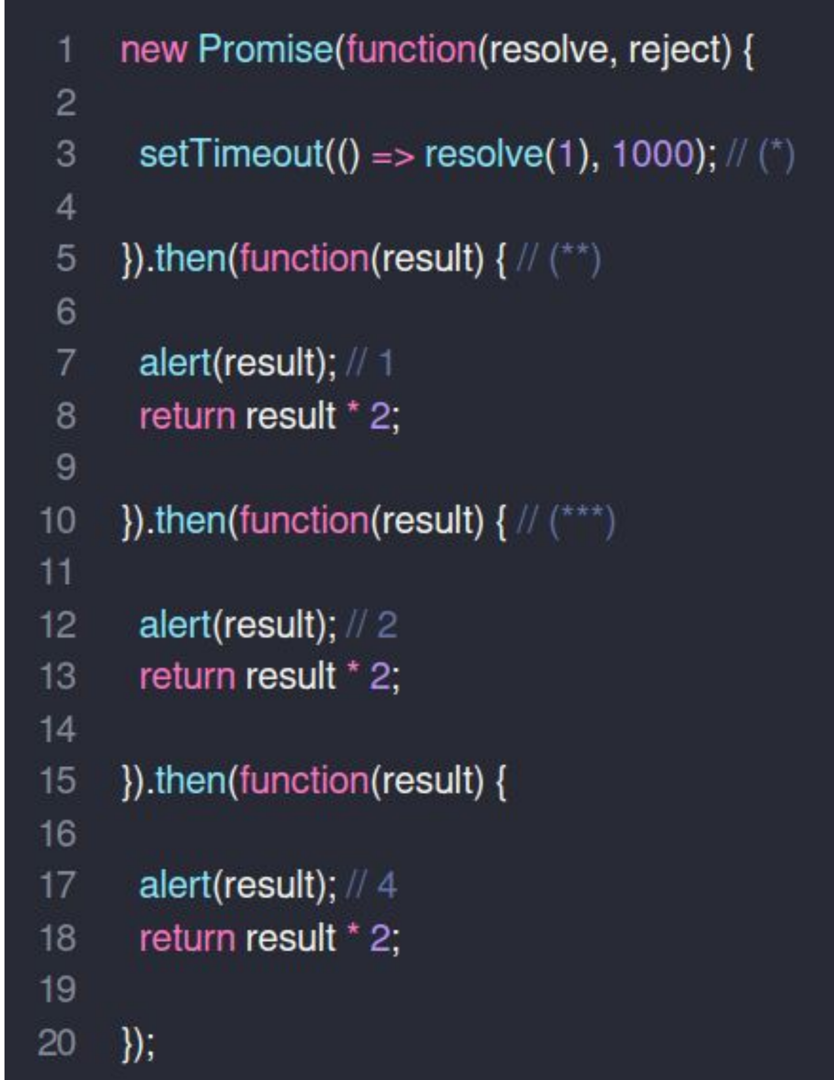


```
new Promise(function(resolve, reject) {

  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;

});
```

Simplified Promise Chaining Example

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.



```
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;

});
```

Returning Promises

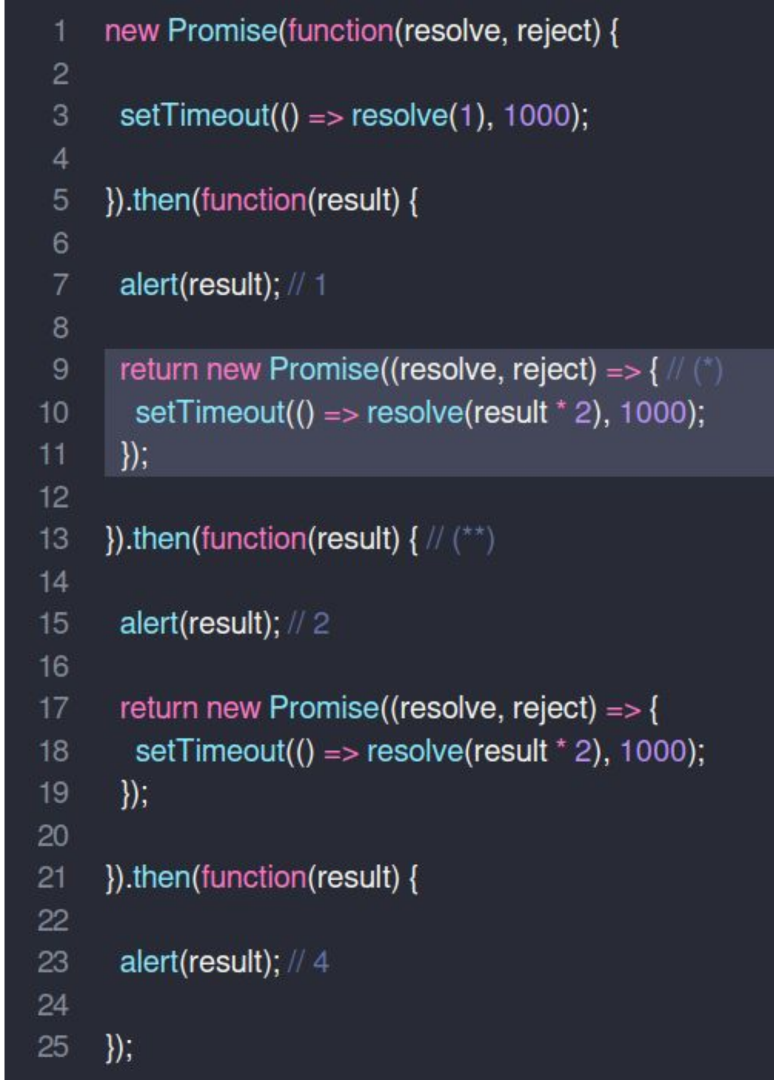
A handler, used in .then(handler) may create and return a promise.

In that case further handlers wait until it settles, and then get its result.

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.

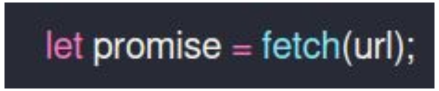


```
new Promise(function(resolve, reject) {

  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

});
```

Fetch API

In frontend programming, promises are often used for network requests.

We'll use the fetch method to load the information about the user from the remote server

Basic Syntax: 

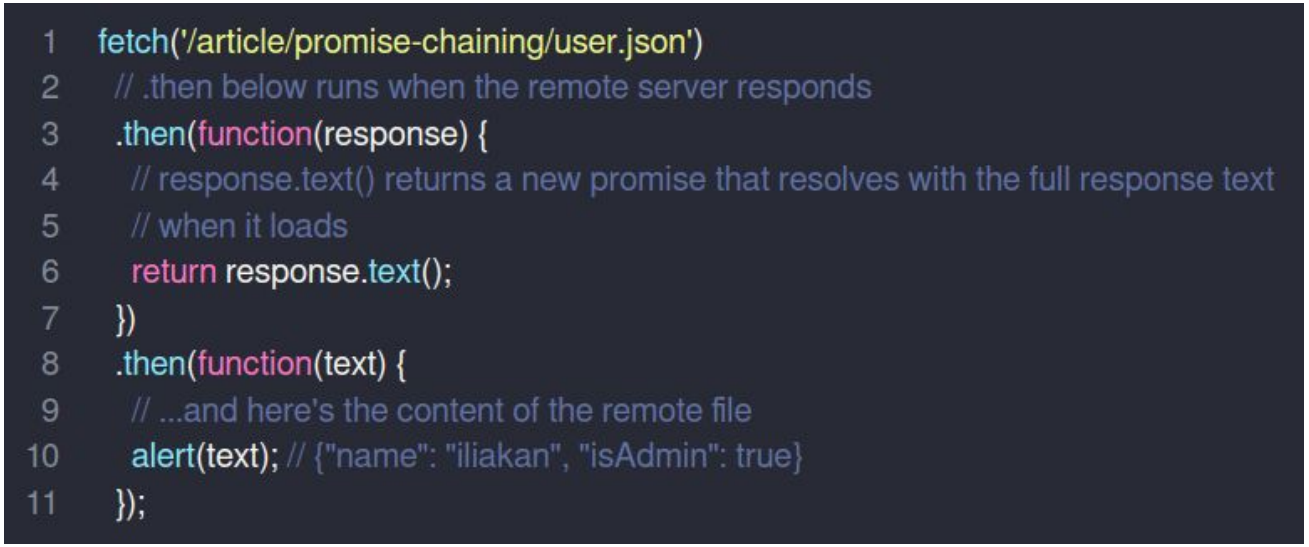
```
let promise = fetch(url);
```

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.

Fetch API

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.



```
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}
  });
```