Asynchronous JS

Synchronous Programming

- In synchronous operations tasks are performed one at a time and only when one is completed, the following is unblocked.
- In other words, you need to wait for a task to finish to move to the next one.
- JavaScript engine executes one line at a time and can not go to execute the next line until the current line execution gets completed.
- Example
- Problem : it is a blocking mode

Asynchronous Programming

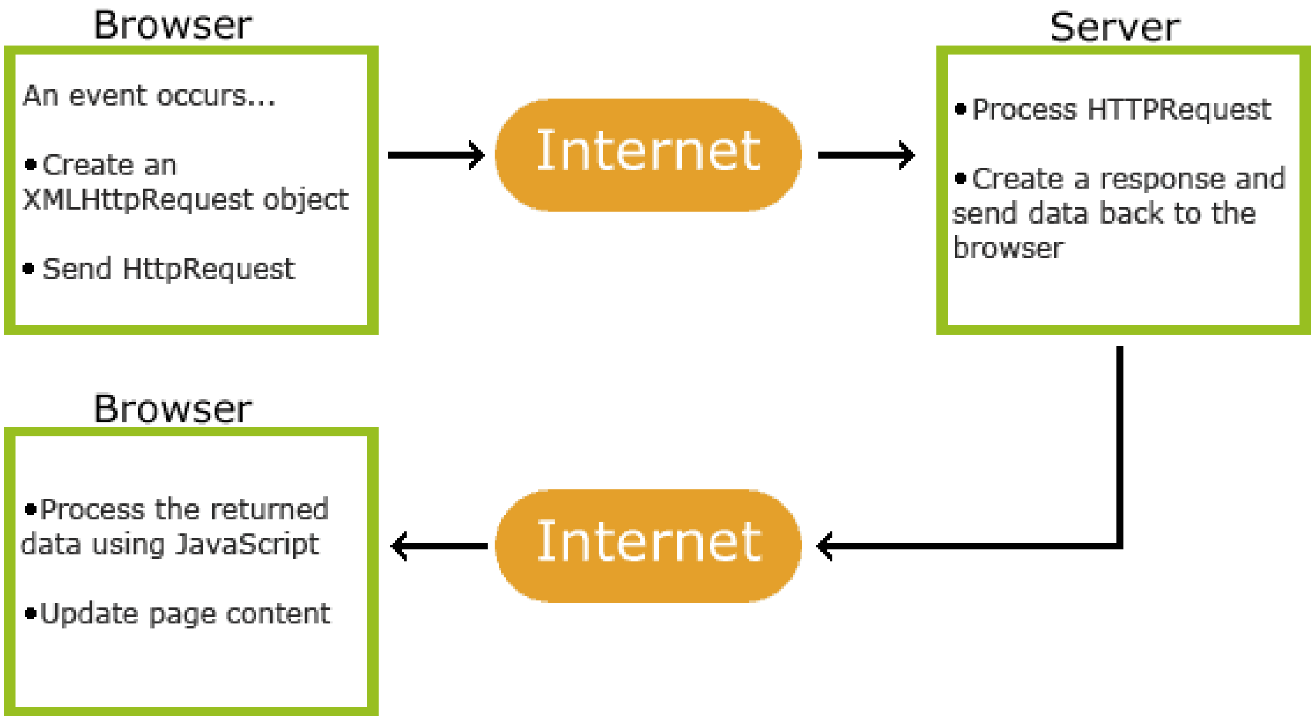
- In asynchronous operations you can move to another task before the previous one finishes.
- This way, with asynchronous programming you're able to deal with multiple requests simultaneously, thus completing more tasks in a much shorter period of time.
- Example

AJAX

- AJAX : **A**synchronous **J**avaScript **A**nd **X**ML
- It is not a programming language.
- It just uses a combination of:
  - A browser built-in XMLHttpRequest object (to request data from a web server)
  - JavaScript and HTML DOM (to display or use the data)
- AJAX applications might use XML to transport data, but it is equally common to transport data as plain text or JSON text.

AJAX

- AJAX allows web pages to be updated asynchronously by exchanging data with a web server behind the scenes.
- This means that it is possible to update parts of a web page, without reloading the whole page.



Browser

An event occurs...

- Create an XMLHttpRequest object
- Send HttpRequest

Internet

Server

- Process HTTPRequest
- Create a response and send data back to the browser

Internet

Browser

- Process the returned data using JavaScript
- Update page content

Callback Hell

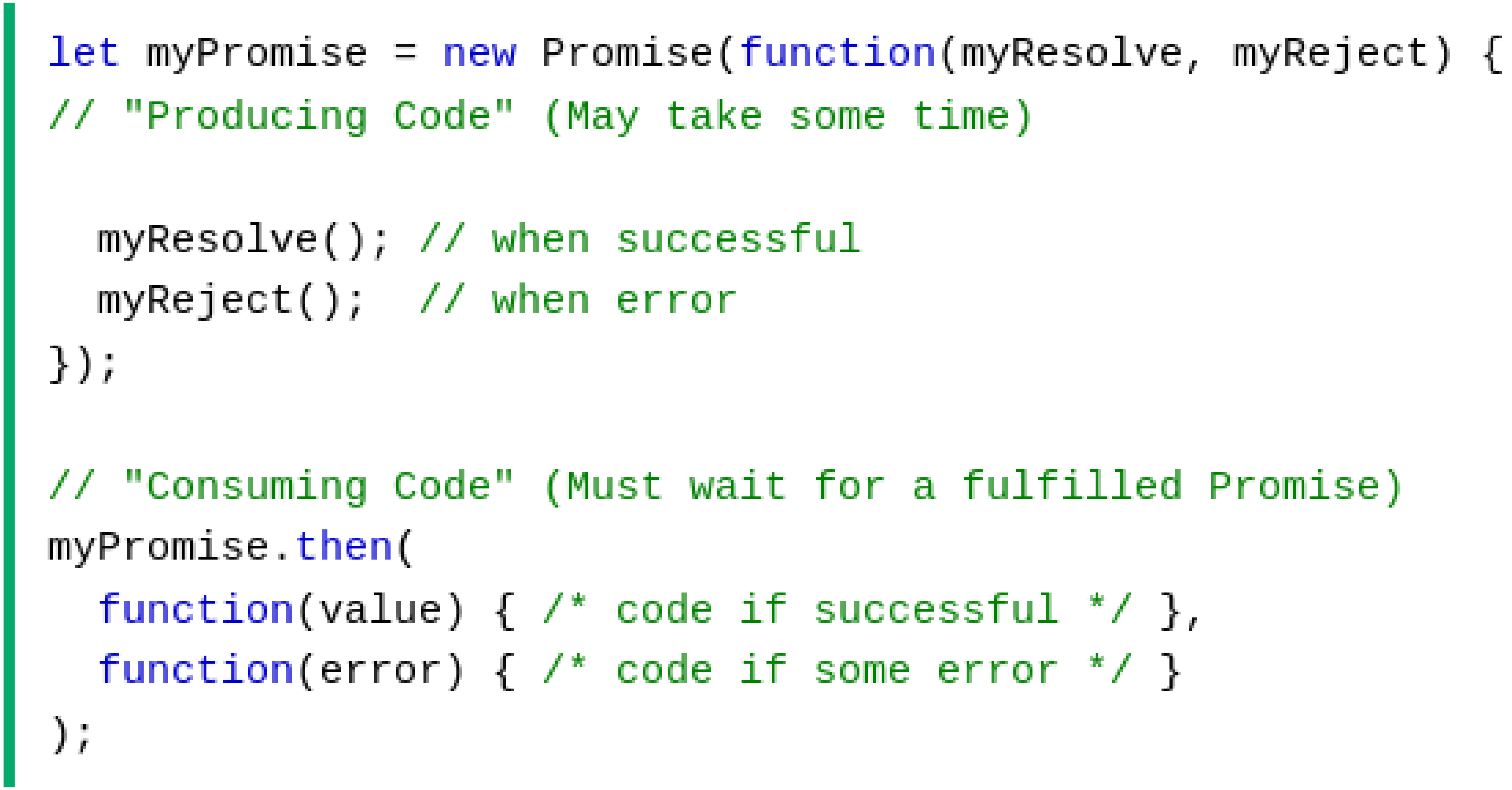
- Callback hell is a phenomenon where multiple callbacks are nested after each other.
- It can happen when you do an asynchronous activity that’s dependent on a previous asynchronous activity.
- These nested callbacks make code much harder to read.
- Example

Handling Callback Hell

- There are two ways to handle the callback hell:
  - Promise
  - async/await

Promise

- Promise is a JavaScript object, which contains both the producing code and calls to the consuming code.



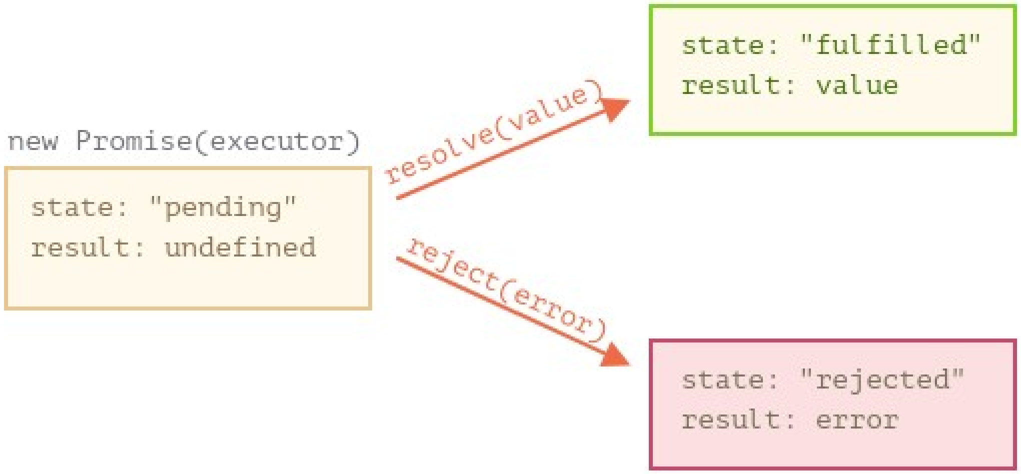
```
let myPromise = new Promise(function(myResolve, myReject) {
// "Producing Code" (May take some time)

  myResolve(); // when successful
  myReject();  // when error
});

// "Consuming Code" (Must wait for a fulfilled Promise)
myPromise.then(
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
```

Promise : Executor Code

- 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.
- Its arguments myResolve and myReject are callbacks provided by JavaScript itself. Our code is only inside the executor.
- When the executor obtains the result, it should call one of these callbacks:
    - **myResolve(value)** — if the job is finished successfully, with result value.
    - **myReject(error)** — if an error has occurred, error is the error object.

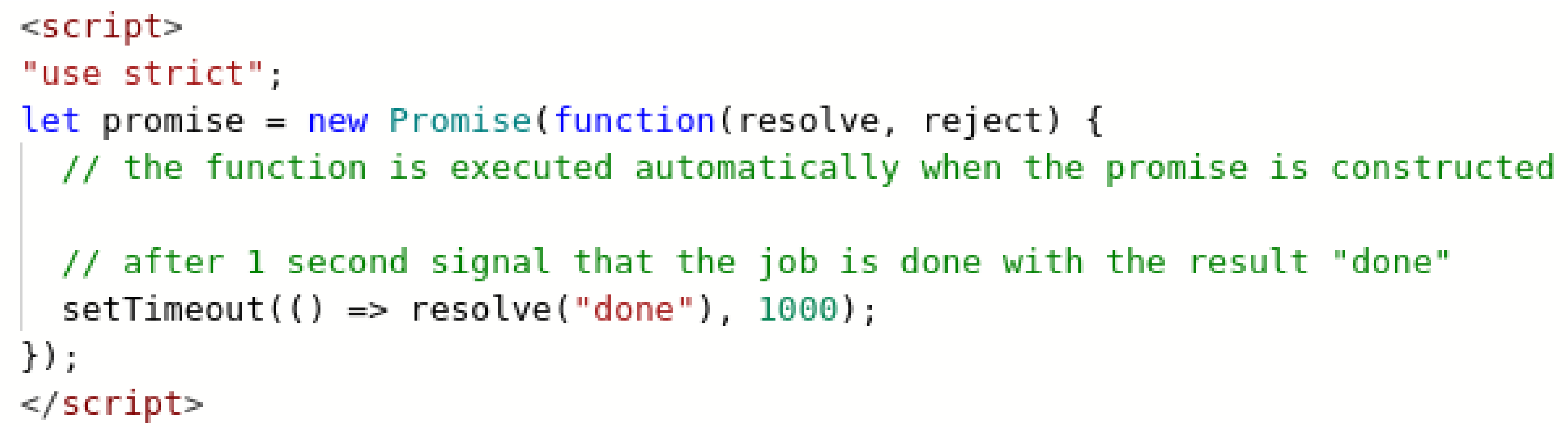
Promise : Properties

- 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) called or **error** when reject(error) is called.
- So the executor eventually moves promise to one of these states:

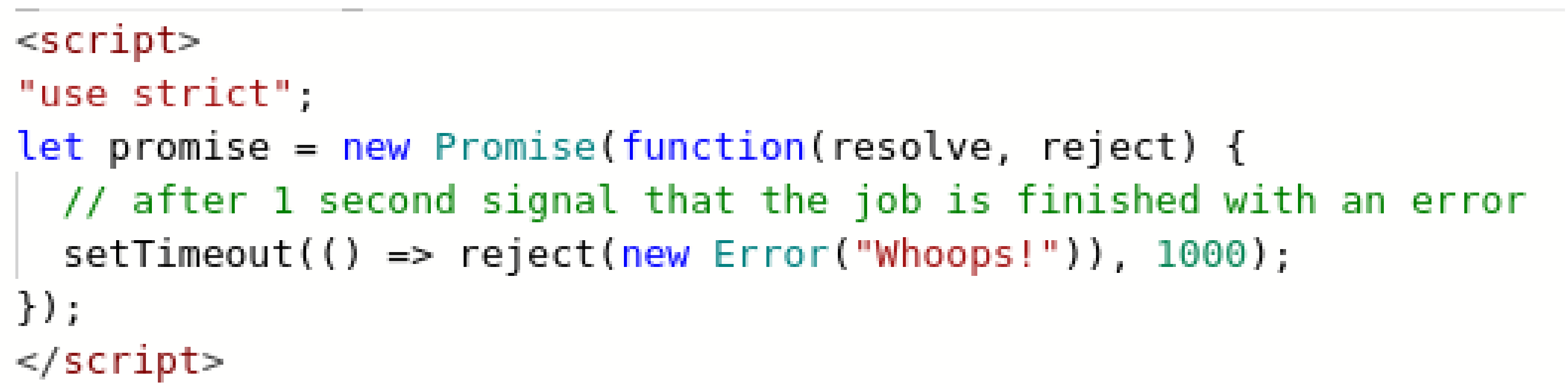


Promise : Producing code example

An example of a promise constructor and a simple executor function with “producing code” that takes time (via setTimeout):



```
<script>
"use strict";
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);
});
</script>
```



```
<script>
"use strict";
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);
});
</script>
```

A promise that is either resolved or rejected is called “settled”, as opposed to an initially “pending” promise.

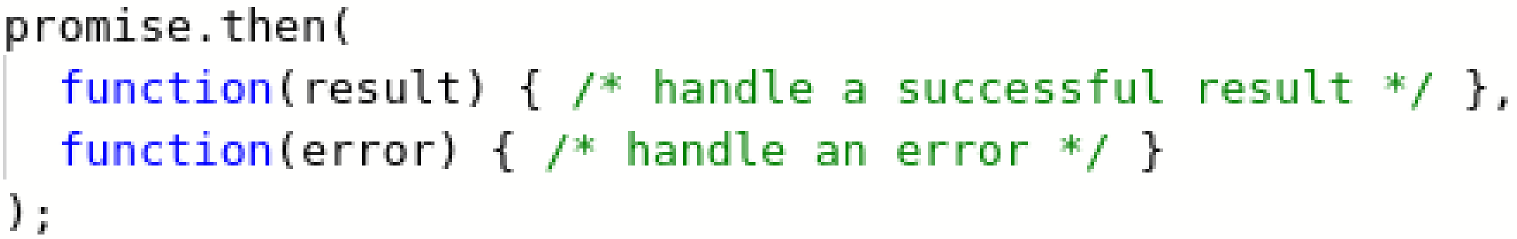
Promise : Consumers

- A Promise object serves as a link between the executor (the “producing code”) and the consuming functions, which will receive the result or error.

- Consuming functions can be registered using methods **.then**, **.catch** and **.finally**.

Consumers: .then

- The Syntax:



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

- The first argument is a function that runs when the promise is resolved, and receives the result.
- The second argument is a function that runs when the promise is rejected, and receives the error.
- Example

Consumers: .catch

- If we’re interested only in errors, then we can use null as the first argument:
  - promise.then(null, errorHandlingFunction)
- Or we can use
  - promise.catch(errorHandlingFunction)
- The call **.catch(f)** is a complete analog of **.then(null, f)**, it’s just a shorthand.
- Example

Consumers: .finally

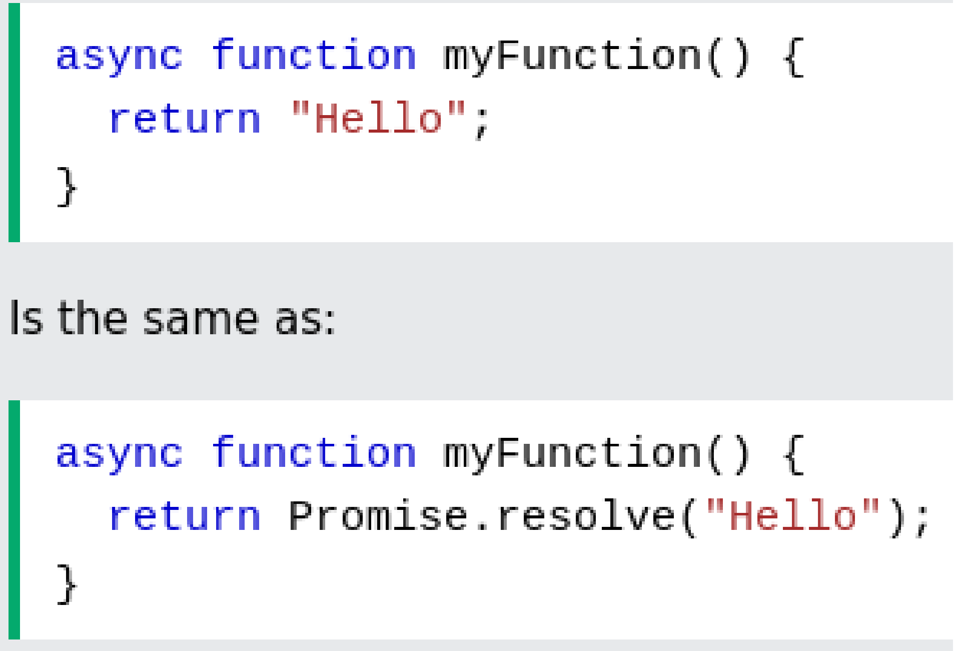
- The call .finally(f) is similar to .then(f, f) in the sense that f always runs when the promise is settled: be it resolve or reject.
- finally is a good handler for performing cleanup, e.g. stopping our loading indicators, as they are not needed anymore, no matter what the outcome is.
- A finally handler 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.
- A finally handler passes through results and errors to the next handler.
- Example

Async and **Await**

- **async** and **await** make promises easier to write.
- async makes a function return a Promise
- await makes a function wait for a Promise

Async

- **Async** Syntax:



```
async function myFunction() {
  return "Hello";
}
```

Is the same as:

```
async function myFunction() {
  return Promise.resolve("Hello");
}
```

- Example

Await

- The keyword await before a function makes the function wait for a promise:
- The await keyword can only be used inside an async function.
- Example