ES6 Day3

By:Nawal Zaki Abdelhady

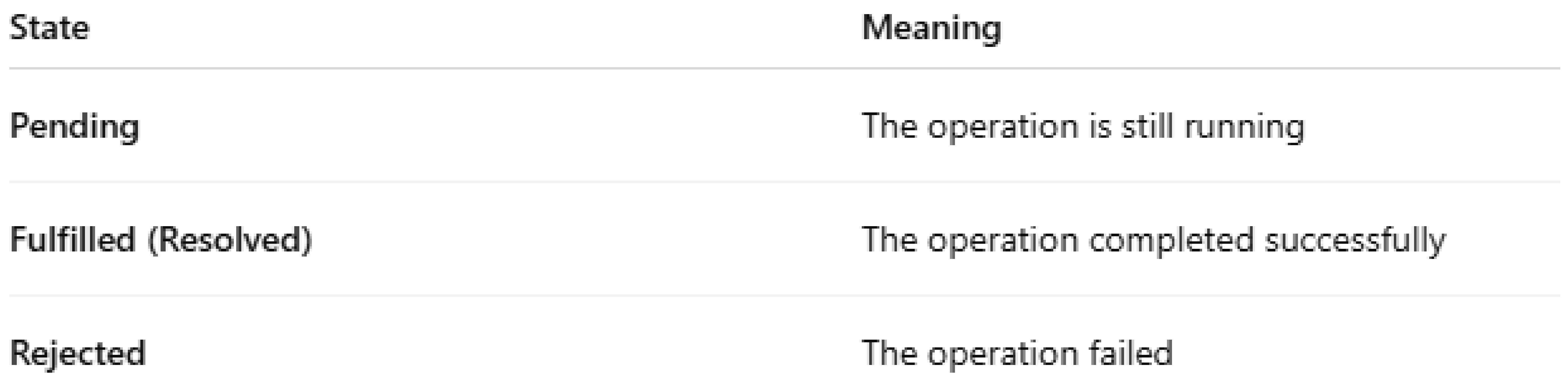
Content

Promises
Async/Await
Fetch
Modules
Observable

Promises

A Promise is an object that represents the result of an asynchronous operation that will be completed in the future.
Instead of using nested callbacks, Promises provide a clean and readable way to handle async tasks.

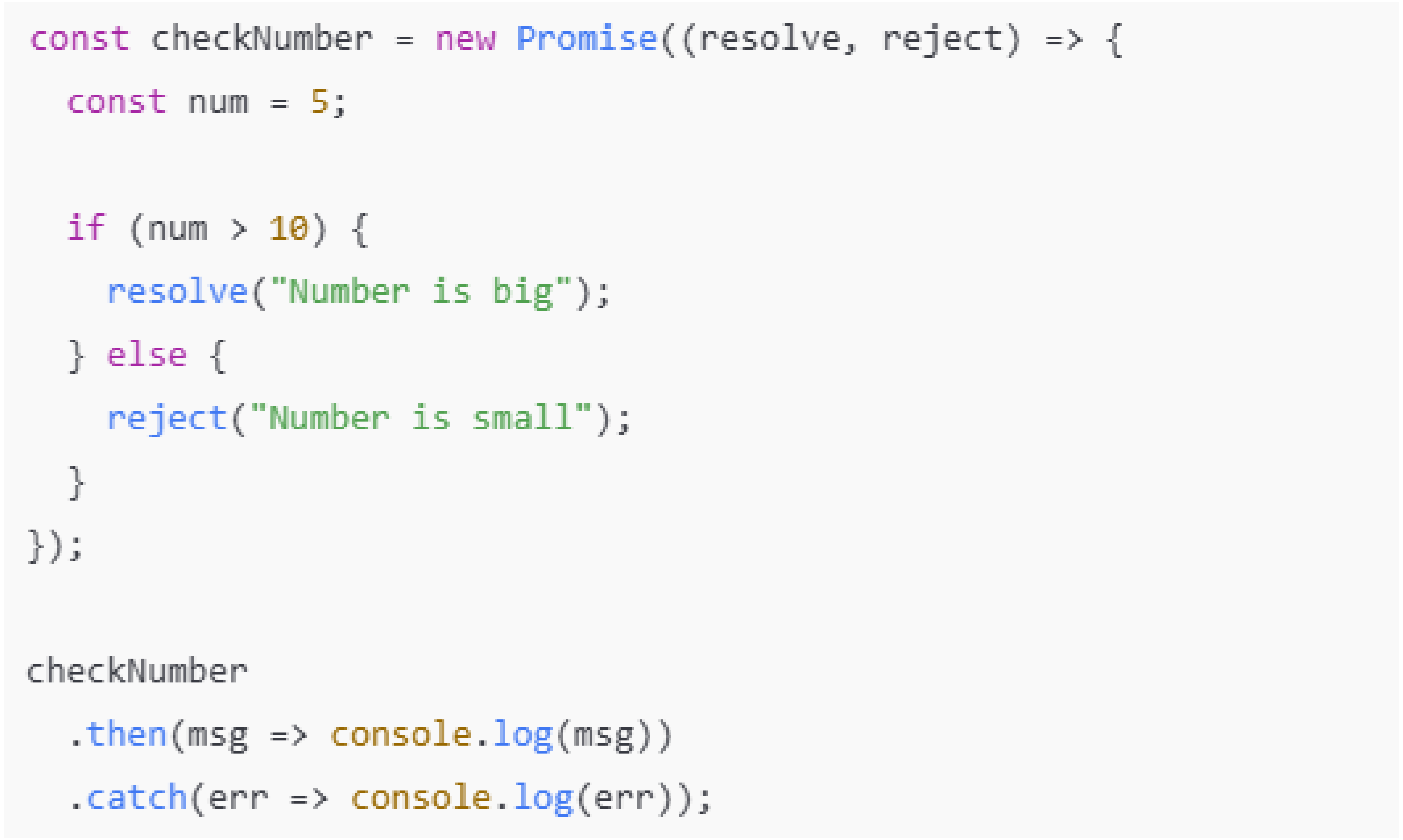
The Event Queue (or Task Queue) stores asynchronous tasks (like timers, fetch responses, or events) waiting to be executed. Once the Call Stack is empty, the Event Loop moves tasks from the Event Queue to the Call Stack.



State	Meaning
Pending	The operation is still running
Fulfilled (Resolved)	The operation completed successfully
Rejected	The operation failed

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Task completed!");
  }, 2000);
});
```

```
myPromise
  .then(result => {
    console.log(result);
  })
  .catch(error => {
    console.log(error);
  });
```



```javascript
const checkNumber = new Promise((resolve, reject) => {
  const num = 5;

  if (num > 10) {
    resolve("Number is big");
  } else {
    reject("Number is small");
  }
});

checkNumber
  .then(msg => console.log(msg))
  .catch(err => console.log(err));
```

Why Promises Are Important

- Avoid Callback Hell
- Make async code easier to read and manage
- Allow chaining multiple async operations
- Work perfectly with async/await

Async / Await

Async / Await is a modern and cleaner way to work with Promises in JavaScript.
It allows you to write asynchronous code in a style that looks synchronous and easy to read
Instead of using .then() and .catch() with Promises,
you can use:

- async → to define a function that returns a Promise
- await → to wait for a Promise to finish before moving to the next line

async Function

When you add async before a function, it automatically returns a Promise

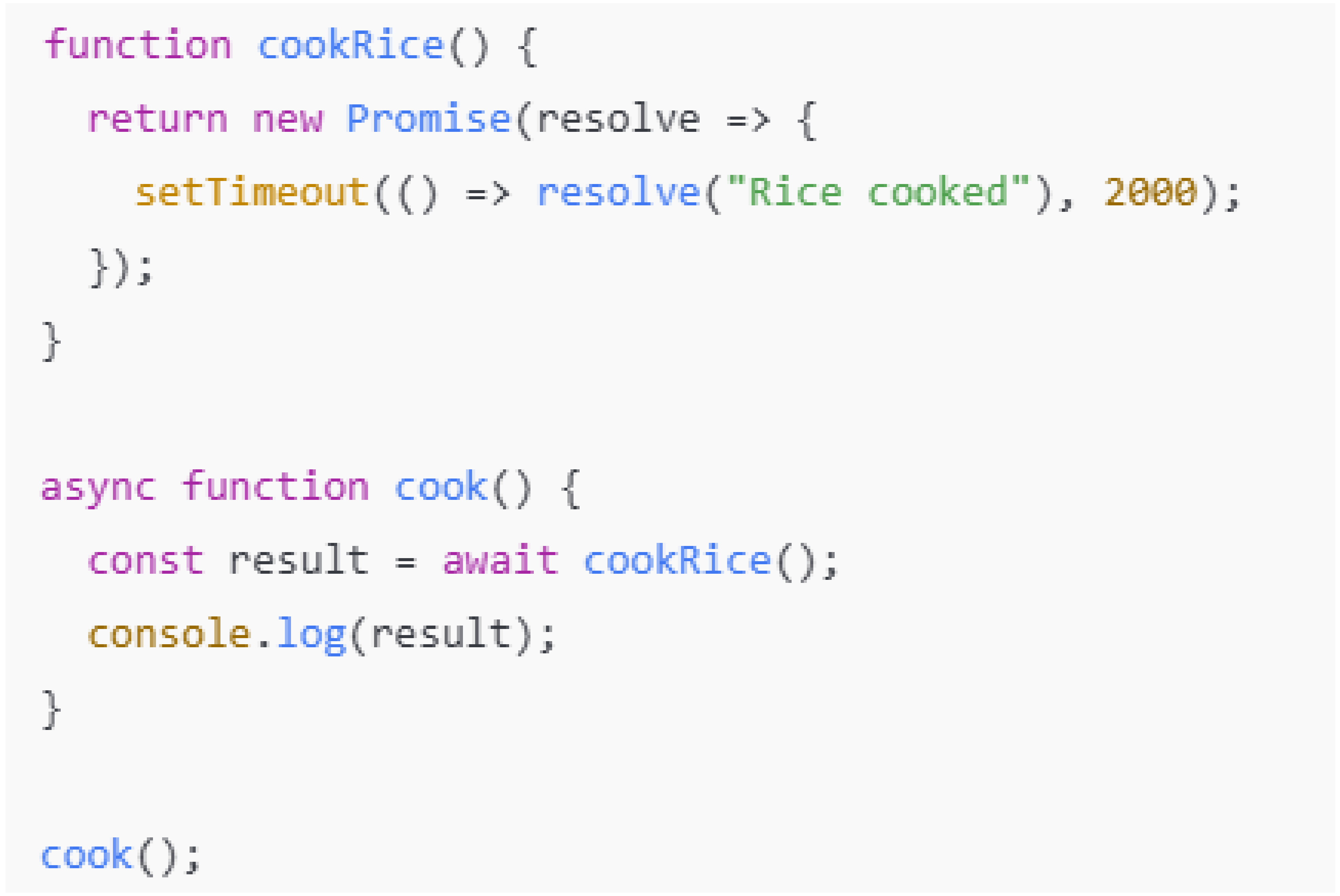
```
async function greet() {
  return "Hello";
}
```

This is the same as:

```javascript
function greet() {
  return Promise.resolve("Hello");
}
```

await Keyword

await pauses the execution of the function until the Promise is resolved.

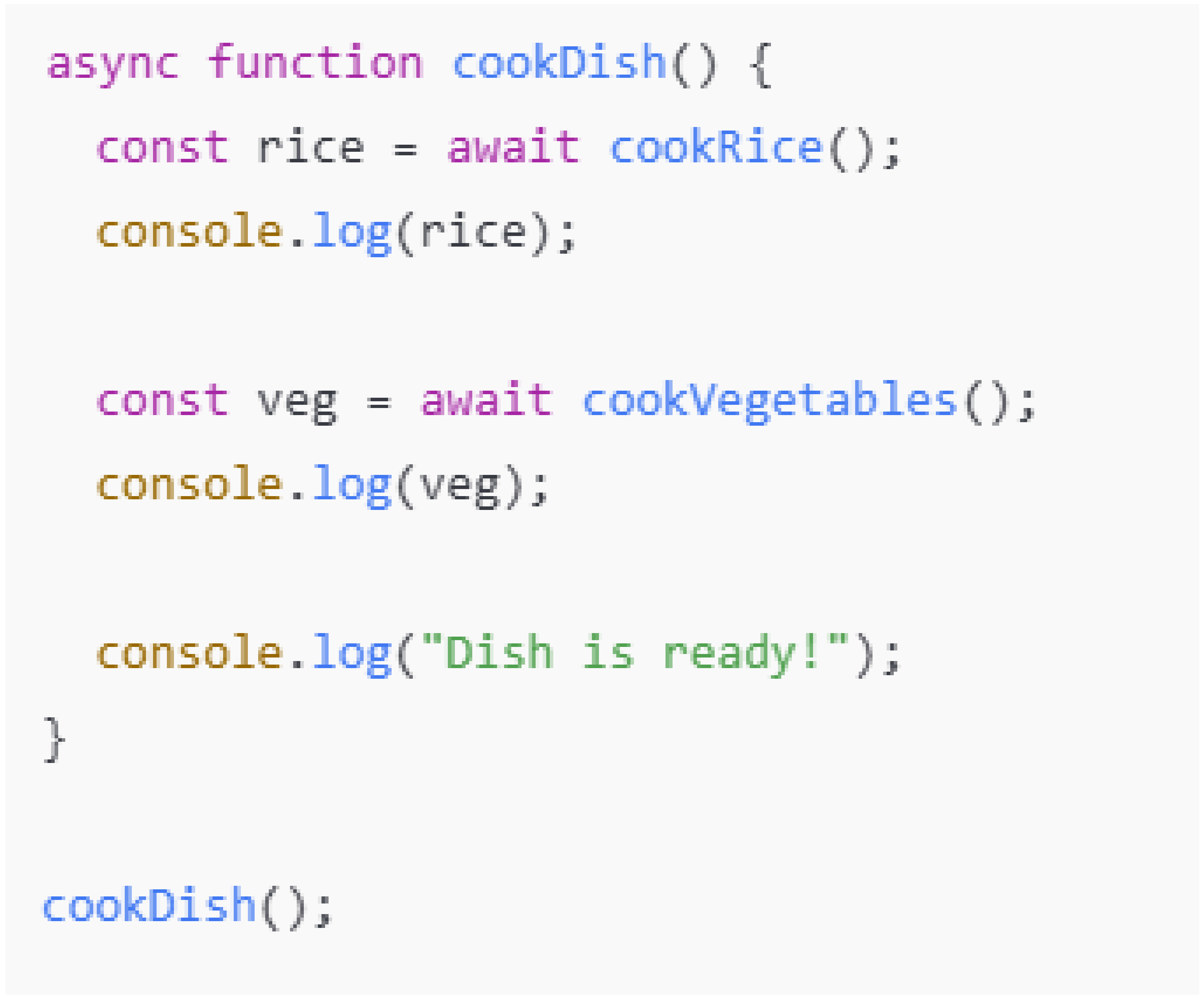


```javascript
function cookRice() {
  return new Promise(resolve => {
    setTimeout(() => resolve("Rice cooked"), 2000);
  });
}

async function cook() {
  const result = await cookRice();
  console.log(result);
}

cook();
```

Example with Try and Catch

```
async function getData() {
  try {
    const response = await fetch("https://api.example.com/data");
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.log("Error:", error);
  }
}
```

The same code, but cleaner and easier to understand:

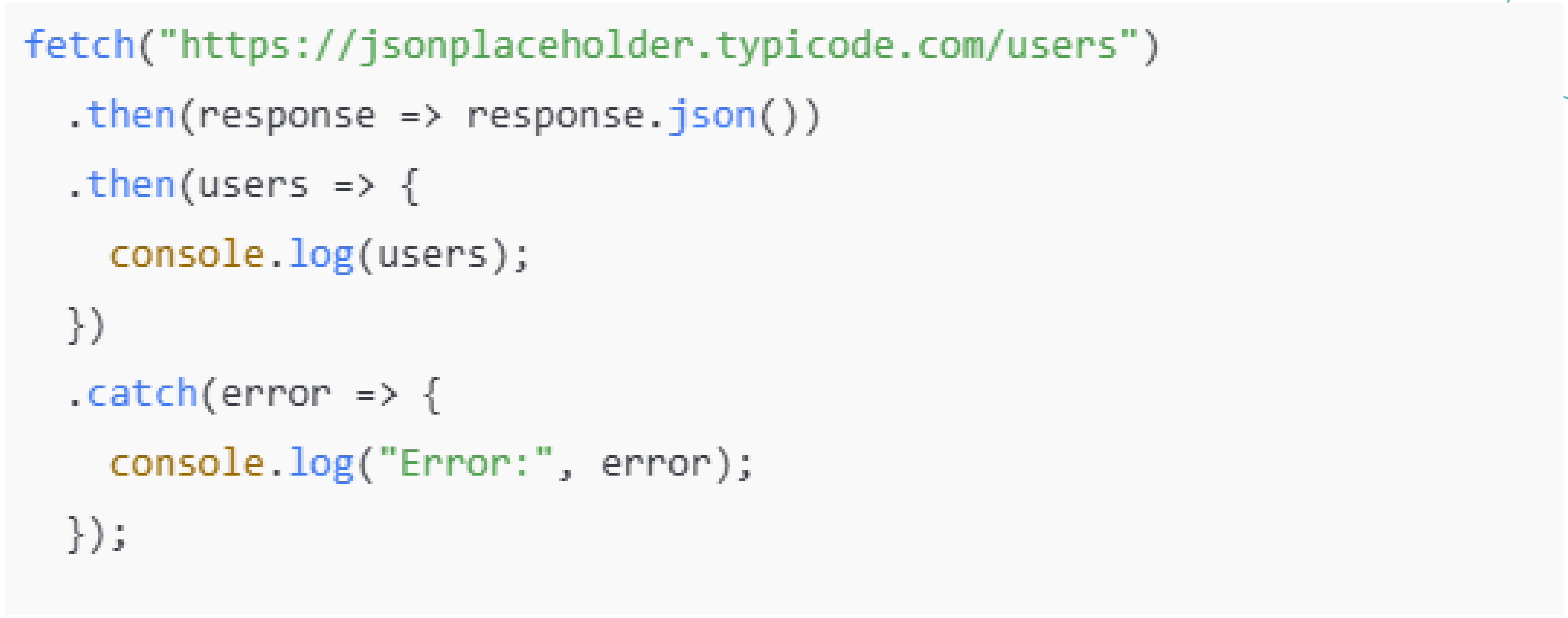


```
async function cookDish() {
  const rice = await cookRice();
  console.log(rice);

  const veg = await cookVegetables();
  console.log(veg);

  console.log("Dish is ready!");
}

cookDish();
```

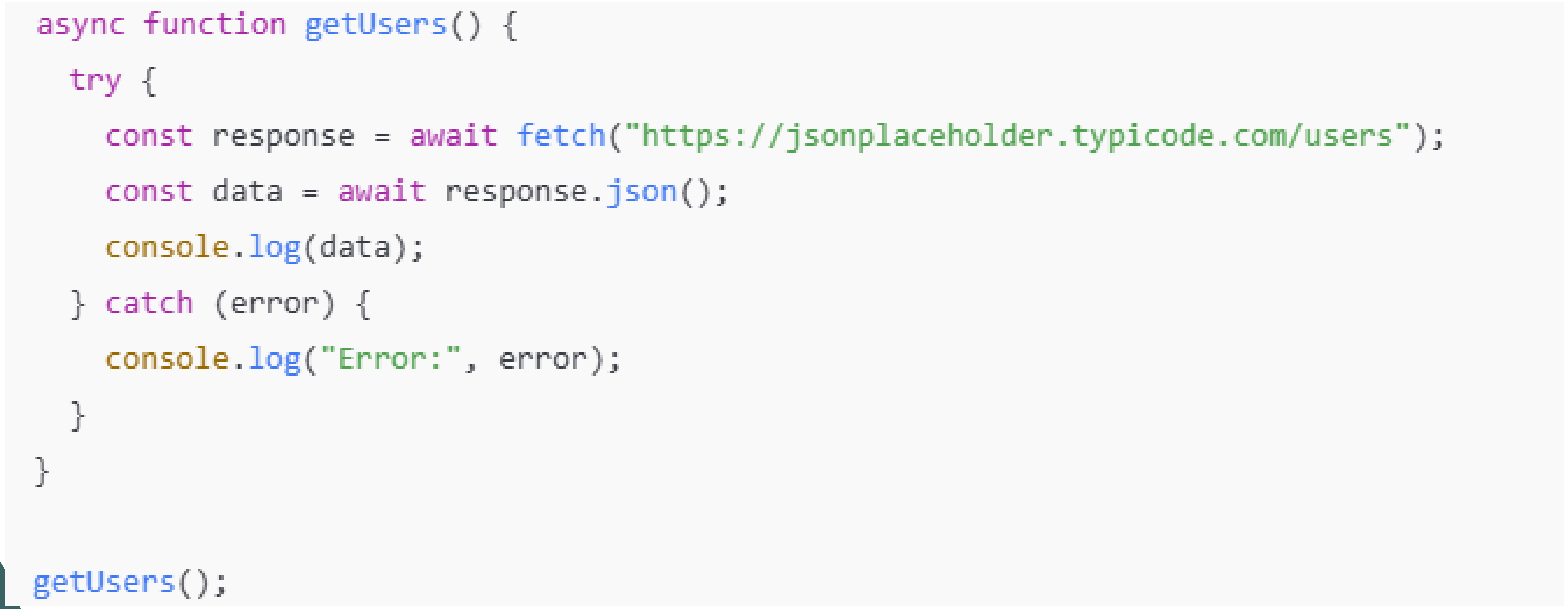
Fetch

fetch is a built-in JavaScript function used to make HTTP requests to a server (API).
It is used to get or send data over the internet and works with Promises.

When you request data from a server, it takes time.
fetch performs this task asynchronously and returns a Promise.



```javascript
fetch("https://jsonplaceholder.typicode.com/users")
  .then(response => response.json())
  .then(users => {
    console.log(users);
  })
  .catch(error => {
    console.log("Error:", error);
  });
```



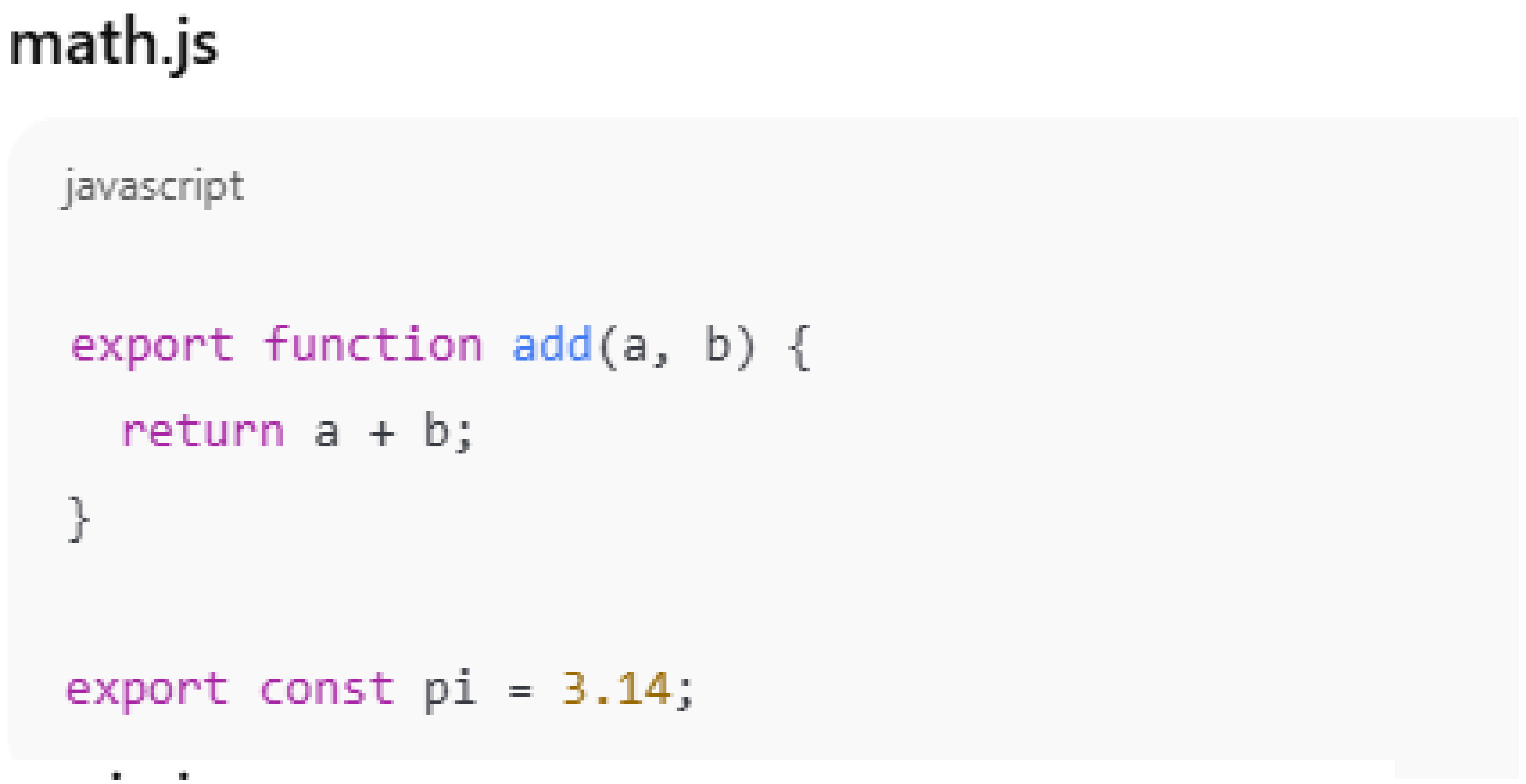
```javascript
async function getUsers() {
  try {
    const response = await fetch("https://jsonplaceholder.typicode.com/users");
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.log("Error:", error);
  }
}

getUsers();
```

```javascript
fetch("https://jsonplaceholder.typicode.com/posts", {
  method: "POST",
  headers: {
    "Content-Type": "application/json"
  },
  body: JSON.stringify({
    title: "JavaScript",
    body: "Learning Fetch API",
    userId: 1
  })
})
  .then(response => response.json())
  .then(data => console.log(data));
```

Modules

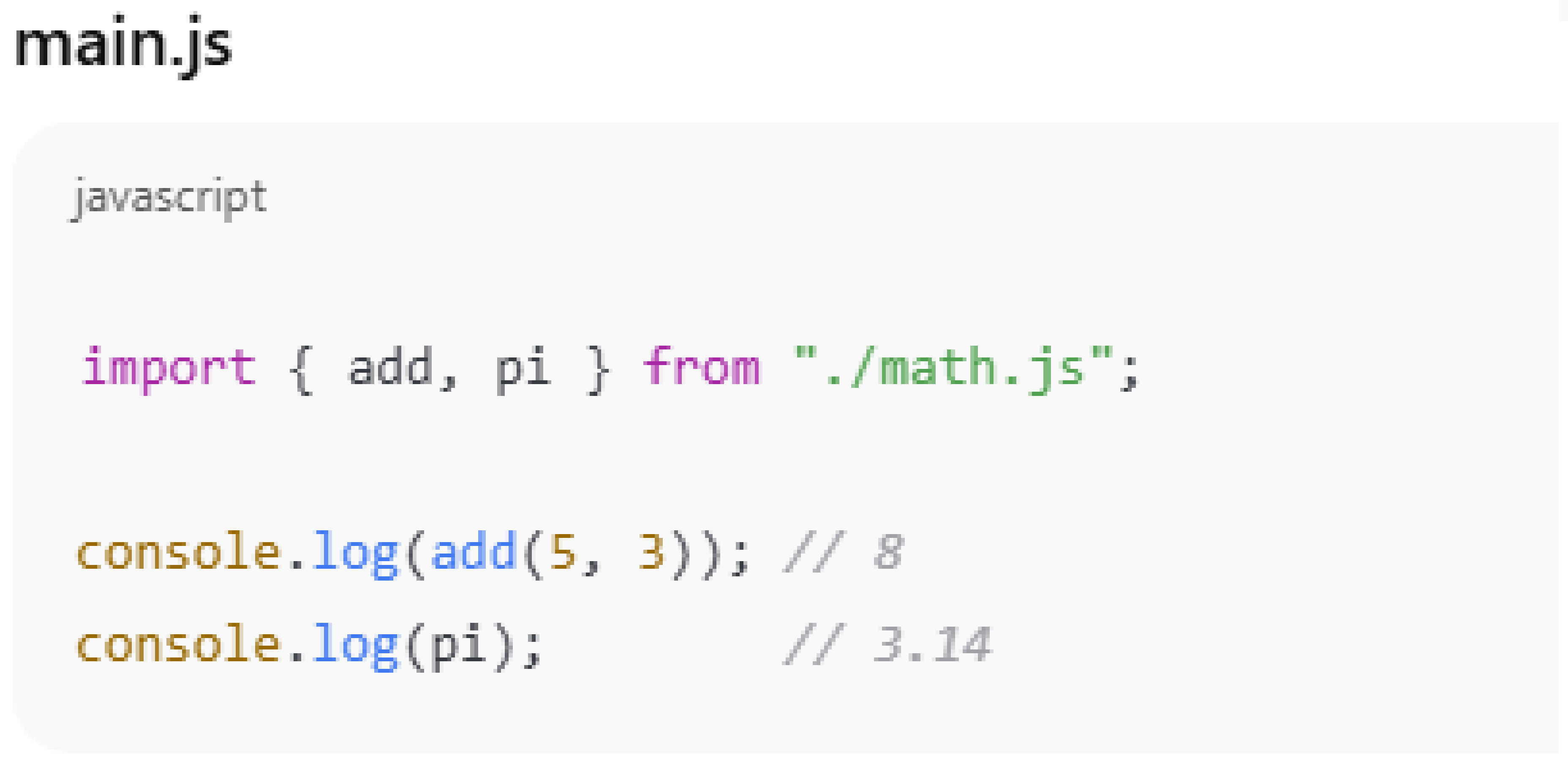
Modules allow you to split your JavaScript code into separate files, where each file has its own functionality.
This helps you organize, reuse, and maintain your code more easily.
Instead of writing all your code in one file, you can create small files (modules) and connect them together.
Each file is treated as a module.
You can:

- Export functions, variables, or classes from one file
- Import them into another file



math.js

```javascript
export function add(a, b) {
  return a + b;
}

export const pi = 3.14;
```



main.js

```javascript
import { add, pi } from "./math.js";

console.log(add(5, 3)); // 8
console.log(pi);        // 3.14
```

Why Modules Are Important

- Better code organization
- Reusable code across projects
- Avoid global variables
- Easier maintenance and debugging
- Used in all modern JavaScript frameworks and applications

Modules work in the browser when using:

```html
<script type="module" src="main.js"></script>
```

Observable

An Observable represents a stream of values over time.
Unlike a Promise, which gives you one value once,
an Observable can give you multiple values continuously.

Promise	Observable
Returns one value	Returns multiple values
Executes once	Can keep emitting values
Cannot be cancelled	Can be cancelled (unsubscribe)
Used for single async result	Used for streams (events, clicks, data)

Observable

Real-Life Examples of Observable

- **Mouse movements**
- **Button clicks**
- **Keyboard typing**
- **Live data from server**
- **Timers**

These are continuous streams of data, not one-time results

Observable

Important Concepts

- Subscribe → start listening to values
- Emit → Observable sends data
- Unsubscribe → stop listening

Used in:

- Angular framework
- Real-time applications
- Event handling
- Live data streams

Generator

A Generator is a special type of function that can pause its execution and resume later from where it stopped.
Generators are defined using function* and use the yield keyword

```javascript
function* countUpTo(n) {
  for (let i = 1; i <= n; i++) {
    yield i;
  }
}

for (const num of countUpTo(5)) {
  console.log(num);
}
```

```javascript
function* myGenerator() {
  yield 1;
  yield 2;
  yield 3;
}
```

```javascript
const gen = myGenerator();

console.log(gen.next()); // { value: 1, done: false }
console.log(gen.next()); // { value: 2, done: false }
console.log(gen.next()); // { value: 3, done: false }
console.log(gen.next()); // { value: undefined, done: true }
```

Thank You