A **Promise** in JavaScript is an object that represents the `eventual completion` (or failure) of an asynchronous operation and its resulting value. Promises are used to handle asynchronous operations more effectively than traditional callback functions, providing a cleaner and more manageable way to deal with code that executes asynchronously, such as API calls, file I/O, or timers.

Using a function that returns a promise

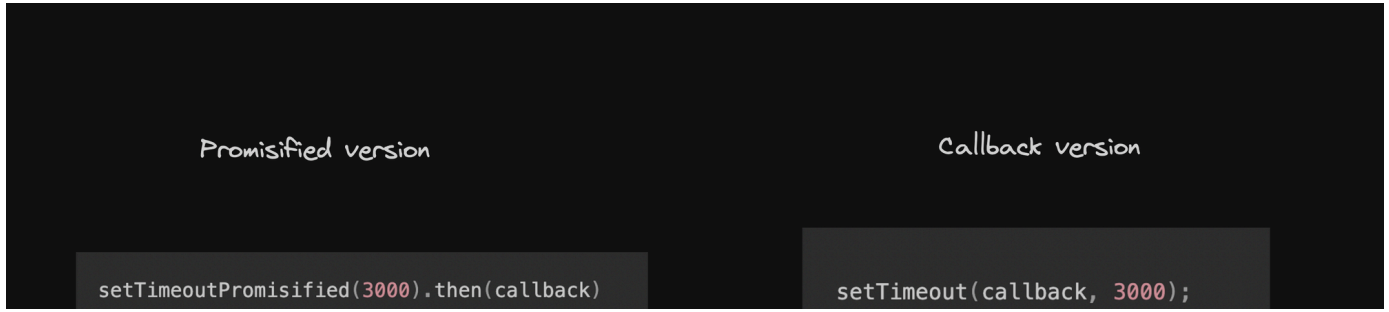
Ignore the function definition of `setTimeoutPromisifed` for now

```
function setTimeoutPromisified(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

function callback() {
    console.log("3 seconds have passed");
}

setTimeoutPromisified(3000).then(callback)
```

Copy



Callback hell

Q: Write code that

1. logs `hi` after 1 second
2. logs `hello` 3 seconds after `step 1`
3. logs `hello there` 5 seconds after `step 2`

▼ Solution (has callback hell)

```
setTimeout(function () {
  console.log("hi");
  setTimeout(function () {
    console.log("hello");

    setTimeout(function () {
      console.log("hello there");
    }, 5000);
```

Copy

```
  }, 3000);
}, 1000);
```

▼ Alt solution (doesnt really have callback hell)

```
function step3Done() {
  console.log("hello there");
}

function step2Done() {
  console.log("hello");
  setTimeout(step3Done, 5000);
}

function step1Done() {
  console.log("hi");
  setTimeout(step2Done, 3000);
}

setTimeout(step1Done, 1000);
```

Copy

Promisified version

Now use the `promisified` version we saw in the last slide

```
function setTimeoutPromisified(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}
```

Copy

▼ Solution #1 (has callback hell)

```
function setTimeoutPromisified(ms) {
  return new Promise((resolve) => setTimeout(resolve, ms));
}

setTimeoutPromisified(1000).then(function () {
  console.log("hi");
  setTimeoutPromisified(3000).then(function () {
    console.log("hello");
    setTimeoutPromisified(5000).then(function () {
      console.log("hello there");
    });
  });
});
```

Copy

▼ Alt solution

```
setTimeoutPromisified(1000)
  .then(function () {
    console.log("hi");
    return setTimeoutPromisified(3000);
  })
  .then(function () {
    console.log("hello");
```

Copy

```
    return setTimeoutPromisified(5000);
  })
  .then(function () {
    console.log("hello there");
  });
```

Async await syntax

The `async` and `await` syntax in JavaScript provides a way to write asynchronous code that looks and behaves like synchronous code, making it easier to read and maintain.

It builds on top of Promises and allows you to avoid chaining `.then()` and `.catch()` methods while still working with asynchronous operations.

`async/await` is essentially syntactic sugar on top of Promises.

Assignment

Write code that

1. logs `hi` after 1 second
2. logs `hello` 3 seconds after `step 1`

3. logs `hello there` 5 seconds after `step 2`

```
function setTimeoutPromisified(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

async function solve() {
    await setTimeoutPromisified(1000);
    console.log("hi");
    await setTimeoutPromisified(3000);
    console.log("hello");
    await setTimeoutPromisified(5000);
    console.log("hi there");
}

solve();
```

Copy

Things to keep in mind

1. You can only call `await` inside a function if that function is `async`
2. You cant have a `top level await`

Defining your own async function

Q: Write a function that

1. Reads the contents of a file
2. Trims the extra space from the left and right
3. Writes it back to the file

1. Callback approach

In the callback approach, the function signature should look something like this -

```
function onDone() {
    console.log("file has been cleaned");
}
cleanFile("a.txt", onDone)
```

Copy

▼ Solution

```
const fs = require("fs");
function cleanFile(filePath, cb) {
  fs.readFile(filePath, "utf-8", function (err, data) {
    data = data.trim();
    fs.writeFile(filePath, data, function () {
      cb();
    });
  });
}
```

Copy

```
function onDone() {
  console.log("file has been cleaned");
}
cleanFile("a.txt", onDone);
```

2. Promisified approach

In the promisified approach, the function signature should look something like this -

```
async function main() {
   await cleanFile("a.txt")
   console.log("Done cleaning file");
}

main();
```

Copy

▼ Solution

```
const fs = require("fs");
function cleanFile(filePath, cb) {
  return new Promise(function (resolve) {
    fs.readFile(filePath, "utf-8", function (err, data) {
      data = data.trim();
      fs.writeFile(filePath, data, function () {
        resolve();
      });
    });
  });
```

Copy

```
}

async function main() {
  await cleanFile("a.txt");
  console.log("Done cleaning file");
}

main();
```

err first callback vs rejects in promises

Callbacks

`fs.readFile` function used an `err first callback` approach to propagate back errors

```
const fs = require("fs")
function afterDone(err, data) {
```

Copy

```
  if (err) {
    console.log("Error while reading file");
  } else {
    console.log(data)
  }
}

fs.readFile("a.txt", "utf-8", afterDone);
```

Promises

Promises use the `reject` argument to propagate errors

```
const fs = require("fs");

function readFilePromisified(filePath) {
  return new Promise(function (resolve, reject) {
    fs.readFile(filePath, "utf-8", function (err, data) {
      if (err) {
        reject("Error while reading file");
      } else {
        resolve(data);
      }
    });
  });
}

function onDone(data) {
  console.log(data);
}
```

Copy

```
function onError(err) {
  console.log("Error: " + err);
}

readFilePromisified("a.txt").then(onDone).catch(onError);
```

Assignments