**Introduction**

Async java script allows to run certain parts of the code in the background without blocking the rest of the functionality.

JS is a synchronous language, which means it will execute one line after the other.

In the example below, you can see that JS executes the code in the same order it was written. And because the function **boilingWater** takes quite some time to execute, the “I’m third” line will take a long time to me executed as well, because it has to wait for the **boilingWater** to finish, which is inefficient.

|  |  |
| --- | --- |
| console.log(`i'm first`);  boilingWater();  console.log(`i'm third`);  function boilingWater() {    console.log('boiling...');    for (let i = 0; i < 10000; i++) {      console.log('still not done...');    }    console.log('done.');  } |  |

**setTimeout**

One way to do async programming is by using **setTimeout**. **This is a browser function, no a JS function, thus it can be executed in parallel with JS.**

In the example below, the browser will see the **setTimeout** function was called and will do execute whatever is in the callback function, while JS will continue doing whatever it is doing. As you can see, it starts the function first because that is the order in which it was written. Then it runs JS and when it finishes the **setTimeout** callback function, if JS is available, it logs “done”.

|  |  |
| --- | --- |
| boilWater(10000);  console.log(`I'm first`);  function boilWater(time) {    console.log('boiling...');    setTimeout(() => {      console.log('done.');    }, time);  } |  |

If JS is busy, like in the code below, it will wait for an opportunity to run the callback function, which in this case is when the for loop ends. This is because the **setTimeout** function case a **minimum** **time**, and **not the exact time**.

|  |  |
| --- | --- |
| boilWater(1000);  console.log(`I'm third`);  for(let i = 0; i<10000;i++){    console.log('still busy')  }  function boilWater(time) {    console.log('boiling...');    setTimeout(() => {      console.log('done.');    }, time);  } |  |

**Callback Hell**

Imagine we have 3 headings and we want to change their color after waiting one second (one heading at a time).

|  |  |
| --- | --- |
|  |  |

We can do that with nested callback functions using setTimeout. At first, you would think that the solution would look time the code below, but no. As it is, everything will change color after 1 second.

const heading1 = document.querySelector('.one');

const heading2 = document.querySelector('.two');

const heading3 = document.querySelector('.three');

const btn = document.querySelector('.btn');

btn.addEventListener('click', () => {

  setTimeout(() => {heading1.style.color = 'red'}, 1000);

  setTimeout(() => {heading2.style.color = 'green'},1000);

  setTimeout(() => {heading3.style.color = 'blue'}, 1000);

});

To make them wait for each other, we need to set callbacks within the callbacks.

// callbacks, promises, async/await

const heading1 = document.querySelector('.one');

const heading2 = document.querySelector('.two');

const heading3 = document.querySelector('.three');

const btn = document.querySelector('.btn');

btn.addEventListener('click', () => {

  setTimeout(() => {

    heading1.style.color = 'red';

    setTimeout(() => {

      heading2.style.color = 'green';

      setTimeout(() => {

        heading3.style.color = 'blue';

      }, 1000);

    }, 1000);

  }, 1000);

});

The problem with this approach is that it becomes **very hard to read** once we have more lines of code (here we just have one, so it kinda looks simple). So let’s ahve a look at other appreaches suck as promises and async/await.

**Promises**

In order to create Promises, we need to use the **new** keyword and use the built-in **Promise** constructor. This promise constructor has a **callback function** as an argument, with **two** **paramenters** (which can have any name).

const promise = new Promise((resolve, reject) => {});

A JavaScript Promise object have 3 states:

* **Pending**: While a Promise object is "pending" (working), the **result is undefined**.
* **Fulfilled**: When a Promise object is "filfilled", the **result is a value**.
* **Rejected**: When a Promise object is "rejected", the **result is an error object**.

In the example below, the state changes depending if value is **true**, **false** or as not been defined yet.

|  |  |
| --- | --- |
| **Promise fulfilled example** | |
| const promise = new Promise((resolve, reject) => {    let value = true;    if (value) {      resolve('value is true');    } else {      reject(`there was a error, value is false`);    }  });  console.log(promise); |  |
| **Promise rejected example** | |
| const promise = new Promise((resolve, reject) => {    let value = false;    if (value) {      resolve('value is true');    } else {      reject(`there was a error, value is false`);    }  });  console.log(promise); |  |
| **Promise pending example** | |
| const promise = new Promise((resolve, reject) => {});  console.log(promise); |  |

But how do we access the returned value (which is this case is just a string, but it can be data from a server)? We can do it by calling the promise and using **then**.

**then** takes a callback functions with an argument. That argument will basically be whatever the promise returns if fulfilled (in this case the string ‘hey value is true’).

|  |  |
| --- | --- |
| const promise = new Promise((resolve, reject) => {    let value = true;    if (value) {      resolve('value is true');    } else {      reject(`there was a error, value is false`);    }  });  promise.then((taco) => {console.log(taco);}) |  |

But what if the value is **false**? How can we make it return the error? Simple, we chain it with catch.

|  |  |
| --- | --- |
| const promise = new Promise((resolve, reject) => {    let value = false;    if (value) {      resolve('value is true');    } else {      reject(`there was a error, value is false`);    }  });  promise    .then((taco) => {      console.log(taco);    })    .catch((err) => {      console.log(err);    }); |  |

And this is a basic setup for a **Promise**.

Now if we apply this to the header color changing example. Here we can see that we can chain then, so the actions are executed in sequence, waiting for the prvious one to complete.

In this case, **resolve()** won’t have any arguments because the function doesn’t return anthing, just changes the style of the headings.

We also have a new error object, which will be returned if the promise is rejected, i.e. the element cannot be found.

const heading1 = document.querySelector('.one');

const heading2 = document.querySelector('.two');

const heading3 = document.querySelector('.three');

const btn = document.querySelector('.btn');

btn.addEventListener('click', () => {

  addColor(1000, heading1, 'red')

    .then(() => addColor(1000, heading2, 'green'))

    .then(() => addColor(1000, heading3, 'blue'))

    .catch((err) => console.log(err));

});

function addColor(time, element, color) {

  return new Promise((resolve, reject) => {

    if (element) {

      setTimeout(() => {

        element.style.color = color;

        resolve();

      }, time);

    } else {

      reject(new Error(`There is no such element ${element}`));

    }

  });

}

**Async/Await**

There is even a better solution, which was introduced in ES6 – Async/Await. The keyword **async** **before a function makes the function return a promise**:

// Regular Function Syntax

async function someFuction(){

  await

}

// Arrow Function Syntax

const someFunction = async() => {

  await

}

The **await** keyword can only be used inside an **async** function.

The **await** keyword makes the function pause the execution and wait for a resolved (either fulfilled or rejected) promise before it continues:

See below the equivalent of the previous example, but this time with the new syntax:

const heading1 = document.querySelector('.one');

const heading2 = document.querySelector('.two');

const heading3 = document.querySelector('.three');

const btn = document.querySelector('.btn');

btn.addEventListener('click', async () => {

  await addColor(1000, heading1, 'red');

  await addColor(1000, heading2, 'green');

  await addColor(1000, heading3, 'blue');

});

function addColor(time, element, color) {

  return new Promise((resolve, reject) => {

    if (element) {

      setTimeout(() => {

        element.style.color = color;

        resolve();

      }, time);

    } else {

      reject(new Error(`There is no such element ${element}`));

    }

  });

}

**Try/Catch Blocks**

The **Try/Catch** blocks is a popular syntax used to handle errors, where:

* The functionality is inside the try block
* The error handling is inside the catch block

We can then use **Try/Catch** blocks to **handle the errors**. This **allows for the rest of the code to still run** if what is inside the try block doesn’t run (for example, it’s referencing some variable that doesn’t exist). That way, we don’t ruin the whole application.

const heading1 = document.querySelector('.one');

const heading2 = document.querySelector('.two');

const heading3 = document.querySelector('.three');

const btn = document.querySelector('.btn');

btn.addEventListener('click', async () => {

  try{

    await addColor(1000, heading1, 'red');

    await addColor(1000, heading2, 'green');

    await addColor(1000, heading3, 'blue');

  }

  catch (error) {

    console.log(error);

  }

});

function addColor(time, element, color) {

  return new Promise((resolve, reject) => {

    if (element) {

      setTimeout(() => {

        element.style.color = color;

        resolve();

      }, time);

    } else {

      reject(new Error(`There is no such element ${element}`));

    }

  });

}