Introduction asynchornous JS

Tags	

- Asynchronous programming is technique that is used when you have a long running task, but still want to handle events whilst the task is running

Example of Asynchronous programming:

- Making HTTP requests using `fetch();`
- Accessing a users camera or microphone using `getUserMedia();`
- Asking a user to select files using `showOpenFilePicker();`

What is **Synchronous programming**?

In Synchronous programming, code is run and executed line by line on a single thread, meaning that the browser waits for each line to finihs to do its work before continuing, example:

```
function greeting (name) {
    return `my name is : ${name}`;
}

const name = "Hamza";
const myGreeting = greeting(name);
console.log(myGreeting); // my name is hamza;
```

- Here, `makeGreeting()` is a **synchronous function** because the caller has to wait for the function to finish its work and return a value before the caller can continue.
- Whilst a Long Running synchronous program is running the user cannot do anything else

- The reason for this is that this JavaScript program is *single-threaded*. A thread is a sequence of instructions that a program follows. Because the program consists of a single thread, it can only do one thing at a time: so if it is waiting for our long-running synchronous call to return, it can't do anything else.

What we need is:

1. Start a long running operation by calling a function
2. Have that function start the operation and return immediately
3. Have a function that does not block the main thread for example by starting a new thread (e.g. event handlers)
4. Notify us with the result of the operation when it eventually completes.

XMLHttpRequest

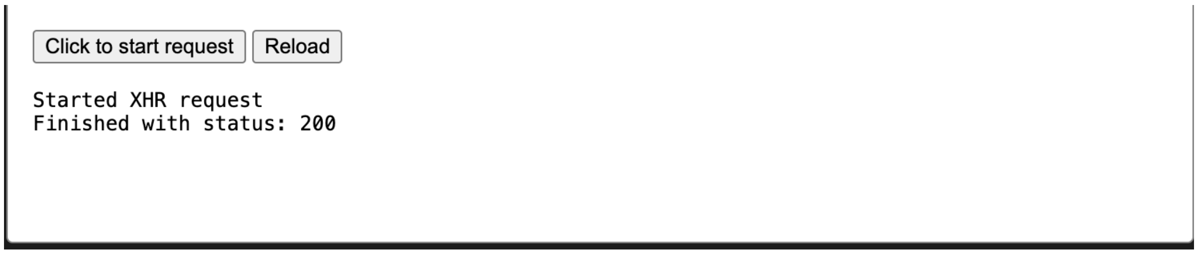
Event handlers are really a form of asynchronous programming: you provide a function (the event handler) that will be called, not right away, but whenever the event happens. If "the event" is "the asynchronous operation has completed", then that event could be used to notify the caller about the result of an asynchronous function call.

Some early asynchronous APIs used events in just this way. The `XMLHttpRequest` API enables you to make HTTP requests to a remote server using JavaScript. Since this can take a long time, it's an asynchronous API, and you get notified about the progress and eventual completion of a request by attaching event listeners to the `XMLHttpRequest` object.

▼ Example Code using XML API

```html
<button id="xhr">Click to start request</button>
<button id="reload">Reload</button>

<pre readonly class="event-log"></pre>
```

```
const log = document.querySelector(".event-log");

document.querySelector("#xhr").addEventListener("click", () =
  log.textContent = "";

  const xhr = new XMLHttpRequest();

  xhr.addEventListener("loadend", () => {
    log.textContent = `${log.textContent}Finished with statu
  });

  xhr.open(
    "GET",
    "https://raw.githubusercontent.com/mdn/content/main/file
  );
  xhr.send();
  log.textContent = `${log.textContent}Started XHR request\n
});

document.querySelector("#reload").addEventListener("click",
  log.textContent = "";
  document.location.reload();
});
```



```
Started XHR request
Finished with status: 200
```

CallBacks

An event handler is a particular type of callback. A callback is just a function that's passed into another function, with the expectation that the callback will be called at the appropriate time. As we just saw, callbacks used to be the main way asynchronous functions were implemented in JavaScript.

However, callback-based code can get hard to understand when the callback itself has to call functions that accept a callback. This is a common situation if you need to perform some operation that breaks down into a series of asynchronous functions. For example, consider the following:

JSCopy to Clipboard

```
function doStep1(init) {
  return init + 1;
}

function doStep2(init) {
  return init + 2;
}

function doStep3(init) {
  return init + 3;
}

function doOperation() {
  let result = 0;
  result = doStep1(result);
  result = doStep2(result);
  result = doStep3(result);
  console.log(`result: ${result}`);
}

doOperation();
```

Here we have a single operation that's split into three steps, where each step depends on the last step. In our example, the first step adds 1 to the input, the second adds 2, and the third adds 3. Starting with an input of 0, the end result is 6 (0 + 1 + 2 + 3). As a synchronous program, this is very straightforward. But what if we implemented the steps using callbacks?

JSCopy to Clipboard

```js
function doStep1(init, callback) {
  const result = init + 1;
  callback(result);
}

function doStep2(init, callback) {
  const result = init + 2;
  callback(result);
}

function doStep3(init, callback) {
  const result = init + 3;
  callback(result);
}

function doOperation() {
  doStep1(0, (result1) => {
    doStep2(result1, (result2) => {
      doStep3(result2, (result3) => {
        console.log(`result: ${result3}`);
      });
    });
  });
}

doOperation();
```

Because we have to call callbacks inside callbacks, we get a deeply nested `doOperation()` function, which is much harder to read and debug. This is sometimes called "callback hell" or the "pyramid of doom" (because the indentation looks like a pyramid on its side).

When we nest callbacks like this, it can also get very hard to handle errors: often you have to handle errors at each level of the "pyramid", instead of having error handling only once at the top level.

For these reasons, most modern asynchronous APIs don't use callbacks. Instead, the foundation of asynchronous programming in JavaScript is the `Promise`, and that's the subject of the next article.

Promises

In JavaScript, a **Promise** 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 like API calls, file system operations, or timers, providing a more manageable approach to handling success and failure than traditional callback functions. Here's a deeper dive into what promises are and how they work:

Key Concepts of Promises

1. **States**: A Promise can be in one of three states:
    - **Pending**: The initial state of a Promise. The operation has not completed yet.
    - **Fulfilled**: The operation completed successfully and the Promise now has a resolved value.
    - **Rejected**: The operation failed and the Promise has a reason for the failure.
2. **Executor Function**: When a Promise is created, it takes an executor function as an argument. This function is executed immediately by the Promise

implementation, passing in two functions as parameters: `resolve` and `reject`. The executor function is responsible for performing the asynchronous operation and calling `resolve` upon successful completion or `reject` if an error occurs.

3. **Thenable**: Promises are "thenable," meaning they have a `then()` method which you can use to schedule actions to be performed after the Promise is settled (either fulfilled or rejected).

```
let myPromise = new Promise(resolve, rejects) {

const condition = true;
if (condition === true) {
    con

}



}
```

In this example, `myPromise` is a Promise object. Depending on the condition, it either resolves with "Success!" or rejects with "Failure." The `then()` method is used to handle the case where the Promise is fulfilled, and the `catch()` method is used to handle the case where the Promise is rejected.

Advantages of Using Promises

- **Chaining**: Promises can be chained, meaning you can perform a sequence of asynchronous operations one after another by linking `then()` calls. Each `then()` receives the result of the previous operation and can return a new value or another Promise.
- **Error Handling**: With promises, you can handle errors at any point in the chain using `catch()`. This simplifies the code by allowing a single error handling section, instead of having to place error checks after each asynchronous operation.

- **Synchronization**: Promises help in synchronizing multiple asynchronous operations using methods like `Promise.all()`, which waits for all promises in an iterable to complete.

Modern JavaScript Asynchronous Handling

Promises are a step towards making asynchronous programming more manageable in JavaScript. They are the basis for newer features like `async` and `await` which use promises under the hood to further simplify asynchronous code with synchronous-style, linear execution flows.

Understanding how to use promises effectively can significantly improve the quality of your JavaScript code, making it easier to handle complex sequences of asynchronous operations with more readable and maintainable code.

Await

- Async function are declared using the keyword `async` at the start of function delceration

```
async function myFunction () {
    //This is an async function
}
```

Inside an async function, you can use the `await` keyword before a call to a function that returns a promise. This makes the code wait at that point until the promise is settled, at which point the fulfilled value of the promise is treated as a return value, or the rejected value is thrown.

This enables you to write code that uses asynchronous functions but looks like synchronous code. For example, we could use it to rewrite our fetch example:

```
async function fetchProducts() {
  try {
    // after this line, our function will wait for the `fetch()` call to be settled
    // the `fetch()` call will either return a Response or throw an error
    const response = await fetch(
      "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
    );
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    // after this line, our function will wait for the `response.json()` call to be settled
    // the `response.json()` call will either return the parsed JSON object or throw an error
    const data = await response.json();
    console.log(data[0].name);
  } catch (error) {
    console.error(`Could not get products: ${error}`);
  }
}

fetchProducts();
```

Here, we are calling `await fetch()`, and instead of getting a `Promise`, our caller gets back a fully complete `Response` object, just as if `fetch()` were a synchronous function!

We can even use a `try...catch` block for error handling, exactly as we would if the code were synchronous.

Note though that async functions always return a promise, so you can't do something like:

```
async function fetchProducts() {
  try {
    const response = await fetch(
      "https://mdn.github.io/learning-area/javascript/apis/fe
tching-data/can-store/products.json",
    );
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    const data = await response.json();
    return data;
  } catch (error) {
    console.error(`Could not get products: ${error}`);
  }
}

const promise = fetchProducts();
console.log(promise[0].name); // "promise" is a Promise objec
t, so this will not work
```

Instead, you'd need to do something like:

```
async function fetchProducts() {
  try {
    const response = await fetch(
      "https://mdn.github.io/learning-area/javascript/apis/fe
tching-data/can-store/products.json",
    );
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    const data = await response.json();
    return data;
  } catch (error) {
    console.error(`Could not get products: ${error}`);
  }
}

const promise = fetchProducts();
promise.then((data) => console.log(data[0].name));
```

Also, note that you can only use `await` inside an `async` function, unless your code is in a JavaScript module. That means you can't do this in a normal script:

```
try {
  // using await outside an async function is only allowed in
a module
  const response = await fetch(
    "https://mdn.github.io/learning-area/javascript/apis/fetc
hing-data/can-store/products.json",
  );
  if (!response.ok) {
    throw new Error(`HTTP error: ${response.status}`);
  }
  const data = await response.json();
  console.log(data[0].name);
} catch (error) {
  console.error(`Could not get products: ${error}`);
}
```

You'll probably use `async` functions a lot where you might otherwise use promise chains, and they make working with promises much more intuitive.