Asynchronous Programming in JavaScript

Asynchronous programming

All programming languages have runtime engines that execute code.

In JavaScript, the runtime engine is single-threaded, so it runs code sequentially (line by line).

Asynchronous programming

Programming languages that are not synchronous are called asynchronous where programs run concurrently.

Although JavaScript is synchronous, we can perform asynchronous programming with it.

Asynchronous programming

Asynchronous programming allows a program to run its tasks concurrently and it makes programs run faster.

Asynchronous programming in JavaScript uses techniques such as Callbacks, Promises, or Async/Await.

Events

An event is something that we can listen for and respond to.

Some objects in JavaScript are event emitters, so we can register event listeners on them.

Events

In the DOM, many HTML elements implement the **EventTarget** interface which provides the **addEventListener** method.

This method accepts a callback function, so whenever the event occurs, the callback is executed.

Events

We can listen for the **click** event on a button.

Every time button is clicked, the text "button was clicked!" will be printed to the console.

```
button.addEventListener('click', () => {
  console.log('button was clicked!');
});
```

Callbacks

A callback is a function used as an argument in another function.

Callbacks allow us to create asynchronous programs in JavaScript by passing the result of a function into another function.

Callbacks

Usually, when the function we call has finished its work, it will execute the callback function with the result.

Callbacks

```
function greet(name) { // 3
    console.log(`Hi ${name}, how do you do?`); // 4
}

function displayGreeting(callback) { // 1
    let name = prompt("Please enter your name"); // 2
    callback(name); //3
};

displayGreeting(greet);  // 1
```

Callbacks

- The **greet** function is used to log a greeting to the console, and it needs the name of the person to be greeted.
- The **displayGreeting** function gets the person's name and has a callback that passes the name as an argument to the **greet** function while calling it.
- The **displayGreeting** function is called with the **greet** function passed to it as an argument.

Imagine a function called **getUsers** which will make a network request to get an array of users.

We can pass a callback function to it, which will be called with this array once network request is complete.

```
console.log('Preparing to get users'); // 1
getUsers(users => {
  console.log('Got users:', users); // 3
});
console.log('Users request sent'); // 2
```

Callbacks

- This example will print "Preparing to get users".
- It calls **getUsers** which will initiate the network request.
- JavaScript doesn't wait for the request to complete.
- It immediately executes the next statement and "Users request sent" will be printed.
- Once the users have been loaded, the callback will be executed and "Got users" will be printed.

Callbacks

If we need to make multiple asynchronous calls in sequence, we’ll end up with nested function calls and callbacks.

Callbacks

Imagine we want to read a file, process some data from that file, then write a new file.

```
readFile('sourceData.json', data => {
  processData(data, result => {
     writeFile(result, 'processedData.json', () => {
        console.log('Done processing');
     });
  });
});
```

Callbacks

In addition, some callback-based APIs can also use error-first callbacks.

These callback functions take two arguments: the first is an error, and the second is the result.

```
readFile('sourceData.json', (error, data) => { // 1
  if (error) {
    console.error('Error reading file:', error); // 2A
    return;
  }

  processData(data, (error, result) => { // 2B
    if (error) {
      console.error('Error processing data:', error); // 3A
      return;
    }
    ...
readFile('sourceData.json', (error, data) => { // 1
    ...
    writeFile(result, 'processedData.json', error => { // 3B
      if (error) {
        console.error('Error writing file:', error); // 4A
        return;
      }

      console.log('Done processing'); // 4B
    });
  });
});
```

Callback hell

Callbacks make it easy to control and make a code asynchronous, but we will run into a problem called callback hell while using them.

This problem arises when we perform multiple asynchronous tasks with callbacks, which might result in nesting callbacks in callbacks.

```
function greet(callback) {
  console.log('Before greet call');  //  1
  setTimeout(function() {
    console.log("Hi Musab");  // 3
    callback();
  }, 1000);
  console.log('After greet call'); // 2
}
function introduce(callback) {
  console.log('Before introduce call'); // 5
  setTimeout(function() {
    console.log("I am your academic advisor"); // 7
    callback();
  }, 1000);
  console.log('After introduce call'); // 6
}
...
```

```
...
function question(callback) {
    console.log('Before question call'); // 9
    setTimeout(function() {
        console.log("Are you currently facing any challenge ..."); // 11
        callback();
    }, 1000);
    console.log('After question call'); // 10
}

greet(function() {  // (1)
    console.log('After greet callback'); // 4
    introduce(function() {
        console.log('After introduce callback'); // 8
        question(function() {
            console.log("Done");  // 12
        });
    });
});
```

Callback hell

Callback functions aren't typically used directly as an asynchronous mechanism in modern APIs.

But, they're the foundation for other types of asynchronous tools such as Promises.

From Callbacks to Promises

We should change the asynchronous programming technique from callbacks to Promises to avoid the callback hell.

From Callbacks to Promises

Most programs consist of a producing code that performs a time-consuming task and a consuming code that needs the result of the producing code.

A Promise links the producing and the consuming code together.

In this example, the **displayGreeting** function is the producing code while the **greet** function is the consuming code.

```
let name;
// producing code
function displayGreeting(callback) {
    name = prompt("Please enter your name");
}
// consuming code
function greet(name) {
    console.log(`Hi ${name}, how do you do?`);
}
```

From Callbacks to Promises

- This code creates a Promise, which takes a function that executes the producing code.
- This function either resolves or rejects its task.
- If the producing code resolves, its result will be passed to the consuming code through the .then handler.

```
let name;

function displayGreeting() { // 3
    name = prompt("Please enter your name");  // 4
}

let promise = new Promise(function(resolve, reject) { // 1
    // producing code
    displayGreeting(); // 3
    resolve(name) // 5
});
...
```

```
...

function greet(result) { // 5
    console.log(`Hi ${result}, how do you do?`); // 6
}

promise.then( // 2
    // consuming code
    result => greet(result), // 5
    error => alert(error)
);
```

From Callbacks to Promises

- We can convert this callback hell's code to Promises by returning a Promise from each function and chaining the function calls together with the .then handler.
- We can also use the .catch handler to catch any error thrown during the execution.

```
function greet() { //  1
    return new Promise(resolve => {  // 2
        setTimeout(function() {
            console.log("Hi Musab"); // 3
            resolve(); // 4
        }, 1000);
    });
}
function introduce() { // 4
    return new Promise(resolve => { // 5
        setTimeout(function() {
            console.log("I am your academic advisor"); // 6
            resolve(); // 7
        }, 1000);
    });
}
...
```

```
...
function question() { // 7
    return new Promise(resolve => {  // 8
        setTimeout(function() {
            console.log("Are you currently facing any challenge..."); // 9
            resolve(); // 10
        }, 1000);
    });
}
greet() // 1
    .then(() => introduce()) // 4
    .then(() => question()) // 7
    .then(() => console.log("Done")) // 10
    .catch(error => console.error("An error occured: ", error));
```

States of a Promise

Any Promise can be in one of these states:

- Pending: This is the initial state and its state while it's still running.
- Fulfilled: This is the state of the Promise when it resolves successfully.
- Rejected: This is the state of the Promise when any error make it not to be resolved.

How to create a Promise

We can create a Promise using the **new** keyword with the **Promise** constructor.

This constructor takes a callback function that takes two arguments, called **resolve** and **reject**.

Each of these arguments is a function provided by the Promise.

How to create a Promise

Inside this callback, we can perform any asynchronous work.

If task is successful, we call the **resolve function** with the final result.

If there was an error, we call the **reject function** with the error.

Get result of a Promise

Often times, we won't actually need to construct a Promise by hand.

We'll typically be working with Promises returned by other APIs.

Get result of a Promise

To get the result of an asynchronous operation:

- We call **then** on the **Promise** object itself.
- It takes a callback function as its argument.
- When the Promise is fulfilled, the callback is executed with the result.

Get result of a Promise

Imagine a function called **getUsers** that asynchronously loads a list of user objects and returns a Promise.

We can get the list of users by calling **then** on Promise returned by **getUsers**.

```
getUsers()
  .then(users => {
    console.log('Got users:', users);
  });
```

Get result of a Promise

This code will continue executing without waiting for the result.

Then, when the users have been loaded, the callback is scheduled for execution.

```
console.log('Loading users'); // 1
getUsers() // 2
  .then(users => { // 4
    console.log('Got users:', users); // 5
  });
console.log('Continuing on'); // 3
```

Get result of a Promise

- In this example, "Loading users" will be printed first.
- The next message that is printed will be "Continuing on", because **getUsers** call is still loading the users.
- Later, we will see "Got users" printed.

What happens if we fail to load the user list?

The **then** function actually takes a second argument, the error handler.

If the Promise is rejected, this callback will be executed with the rejection value.

```
getUsers()
  .then(users => {
    console.log('Got users:', users);
  }, error => {
    console.error('Failed to load users:', error);
  });
```

Get result of a Promise

Since a Promise can only ever be either fulfilled or rejected, but not both, only one of these callback functions will be executed.

Get result of a Promise

What if we need to work with multiple Promises in series?

Consider example where we load some data from a file, do some processing, and write the result to a new file.

Get result of a Promise

```
readFile('sourceData.json')
 .then(data => {
   processData(data)
     .then(result => {
       writeFile(result, 'processedData.json')
         .then(() => {
           console.log('Done processing');
         });
     });
 });
```

Get result of a Promise

We still have the nesting issue that we had with the callback approach.

We can chain Promises together in a flat sequence.

Promise chaining

The idea is that **then** method returns another Promise.

Whatever value we return from the **then** callback becomes the fulfilled value of this new Promise.

Promise chaining

Consider a **getUsers** function that returns a Promise that gets fulfilled with an array of user objects.

This code results in a new Promise that will be fulfilled with the first user object in array.

```
getUsers()
  .then(users => users[0])
  .then(firstUser => {
    console.log('First user:', firstUser.username);
  });
```

Promise chaining

This process of returning a Promise, calling **then**, and returning another value, resulting in another Promise, is called chaining.

Promise chaining

What if, instead of returning a value from **then** handler, we returned another Promise?

Consider again the file-processing example, where these are asynchronous functions that return Promises.

Promise chaining

- The **then** handler calls **processData**, returning the resulting Promise.
- The new Promise will become fulfilled when the Promise returned by **processData** is fulfilled, giving us the same value.
- So the code would return a Promise that will be fulfilled with the processed data.

Promise chaining

We can chain multiple Promises, one after the other, until we get to the final value we need.

```
readFile('sourceData.json')
  .then(data => processData(data))
  .then(result => writeFile(result, 'processedData.json'))
  .then(() => console.log('Done processing'));
```

Promise chaining

This code will result in a Promise that won't be fulfilled until after processed data is written to a file.

"Done processing!" will be printed to the console, and then the final Promise will become fulfilled.

Error handling in Promise chains

In file-processing example, an error can occur at any stage in the process.

We can handle an error from any step in Promise chain by using **catch** method.

Error handling in Promise chains

If one of Promises in chain is rejected, callback function passed to **catch** will execute and rest of chain is skipped.

```
readFile('sourceData.json')
  .then(data => processData(data))
  .then(result => writeFile(result, 'processedData.json'))
  .then(() => console.log('Done processing'))
  .catch(error => console.log('Error while processing:', error));
```

Error handling in Promise chains

We might have some code we want to execute regardless of Promise result.

Maybe we want to close a database or a file.

```
openDatabase()
  .then(data => processData(data))
  .catch(error => console.error('Error'))
  .finally(() => closeDatabase());
```

Wait for all tasks to complete

What if we want to run multiple tasks at the same time, and wait until they all complete?

Wait for all tasks to complete

Promise.all takes an array of Promises, and returns a new Promise.

This Promise will be fulfilled once all of the other Promises are fulfilled.

This fulfillment value will be an array containing fulfillment values of each Promise in the input array.

Wait for all tasks to complete

We have a function **loadUserProfile** that loads a user's profile data, and another function **loadUserPosts** that loads a user's posts.

There is a third function, **renderUserPage**, that needs the profile and list of posts.

Wait for all tasks to complete

```
const userId = 100;

const profilePromise = loadUserProfile(userId);
const postsPromise = loadUserPosts(userId);

Promise.all([profilePromise, postsPromise])
  .then(results => {
    const [profile, posts] = results;
    renderUserPage(profile, posts);
  });
```

Wait for all tasks to complete

If any of Promises passed to **Promise.all** is rejected with an error, the resulting Promise is also rejected with that error.

If any of other Promises are fulfilled, those values are lost.

Async/Await

async and **await** are special keywords that simplify working with Promises.

They remove the need for callback functions and calls to **then** or **catch**.

This effectively pauses execution of function until Promise is fulfilled.

Async/Await

To make a function asynchronous using **async/await**, we have to write the **async** keyword before the function declaration.

Then, we write the **await** keyword before the producing code's execution call.

```
let name;

function displayGreeting() { // 2
    name = prompt("Please enter your name"); // 3
    return name; // 4
}

function greet(result) { // 5
    console.log(`Hi ${result}, how do you do?`); //  6
}

...
```

```
…
async function greeting() { // 1
    // producing code
    let result = // 4
      await displayGreeting(); // 2
    // consuming code
    greet(result); // 5
};

greeting(); // 1
```

Async/Await

- In this example, the producing code is the **displayGreeting** function, and the consuming code is the **greet** function.
- The **greeting** function is the Promise that connects the producing and the consuming code.
- It waits for the result returned from **displayGreeting** function and passes that result to the **greet** function.

How to use async and await

Promise chains can be used, too.

```
const data = await readFile('sourceData.json');
const result = await processData(data);
await writeFile(result, 'processedData.json');
```

How to use async and await

To use the **await** keyword, function must be marked as an async function.

We need to place **async** keyword before the function.

```
async function processData(sourceFile, outputFile) {
  const data = await readFile(sourceFile);
  const result = await processData(data);
  writeFile(result, outputFile);
}
```

How to use async and await

Async functions always implicitly return a Promise.

If we return a value from an async function, the function will actually return a Promise that is fulfilled with that value.

How to use async and await

Since this is an async function, it doesn't return the sum but rather a Promise that is fulfilled with the sum.

```
async function add(a, b) {
  return a + b;
}
add(2, 3).then(sum => {
   console.log('Sum is:', sum);
});
```

Error Handling in Async/Await

We can handle errors that arise when we perform asynchronous operations with **async/await** using **try...catch** statement.

The asynchronous operation executes in the **try** block, and we can handle errors in the **catch** block.

If we are awaiting a Promise, and it is rejected, an error will be thrown.

To handle it, we can put it in a **try-catch** block.

```
try {
    const data = await readFile(sourceFile);
    const result = await processData(data);
    await writeFile(result, outputFile);
} catch (error) {
    console.error('Error occurred while processing:', error);
}
```

Error Handling in Async/Await

```
async function greeting() {
    try {
        let result = await displayGreeting();
        greet(result);
    } catch(err) {
        console.error(err)
    }
};
```