JavaScript Asynchronous Review

- **Synchronous JavaScript** is executed sequentially and waits for the previous operation to finish before moving on to the next one.
- **Asynchronous JavaScript** allows multiple operations to be executed in the background without blocking the main thread.
- **Thread** is a sequence of instructions that can be executed independently of the main program flow.
- **Callback functions** are functions that are passed as arguments to other functions and are executed after the completion of the operation or as a result of an event.

The JavaScript engine and JavaScript runtime

- The **JavaScript engine** is a program that executes JavaScript code in a web browser. It works like a converter that takes your code, turns it into instructions that the computer can understand and work accordingly.
- V8 is an example of a JavaScript engine developed by Google.
- The **JavaScript runtime** is the environment in which JavaScript code is executed. It includes the JavaScript engine which processes and executes the code, and additional features like a web browser or Node.js.

The Fetch API

- The Fetch API allows web apps to make network requests, typically to retrieve or send data to the server. It provides a fetch() method that you can use to make these requests.
- You can retrieve text, images, audio, JSON, and other types of data using the Fetch API.

HTTP methods for Fetch API

The Fetch API supports various HTTP methods to interact with the server. The most common methods are:

- **GET**: Used to retrieve data from the server. By default, the Fetch API uses the GET method to retrieve data.

```
fetch('https://api.example.com/data')
```

To use the fetched data, it must be converted to JSON format using the .json() method:

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
```

In this code, the response coming from the Fetch API is a promise and the .then handler is converting the response to a JSON format.

- **POST**: Used to send data to the server. The POST method is used to create new resources on the server.

```
fetch('https://api.example.com/users', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    name: 'John Doe',
    email: 'john@example.com'
  })
})
```

In this example, we're sending a POST request to create a new user. We have specified the method as POST, set the appropriate headers, and included a body with the data we want to send. The body needs to be a string, so we use JSON.stringify() to convert our object to a JSON string.

- **PUT**: Used to update data on the server. The PUT method is used to update existing resources on the server.

```
fetch('https://api.example.com/users/45', {
  method: 'PUT',
  headers: {
    'Content-Type': 'application/json',
```

```
  },
  body: JSON.stringify({
    name: 'John Smith',
    email: 'john@example.com'
  })
})
```

In this example, we are updating the ID 45 that is specified at the end of the URL. We have used the PUT method on the code and also specified the data as the body which will be used to update the identified data.

- **DELETE**: Used to delete data on the server. The DELETE method is used to delete resources on the server.

```
fetch('https://api.example.com/users/45', {
  method: 'DELETE'
})
```

In this example, we're sending a DELETE request to remove a user with the ID 45.

Promise and promise chaining

- **Promises** are objects that represent the eventual completion or failure of an asynchronous operation and its resulting value. The value of the promise is known only when the async operation is completed.
- Here is an example to create a simple promise:

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Data received successfully');
  }, 2000);
});
```

- The .then() method is used in a Promise to specify what should happen when the Promise is fulfilled, while .catch() is used to handle any errors that occur.
- Here is an example of using .then() and .catch() with a Promise:

```
promise
  .then(data => {
    console.log(data);
  })
  .catch(error => {
    console.error(error);
  });
```

In the above example, the .then() method is used to log the data received from the Promise, while the .catch() method is used to log any errors that occur.

- **Promise chaining**: One of the powerful features of Promises is that we can chain multiple asynchronous operations together. Each .then() can return a new Promise, allowing you to perform a sequence of asynchronous operations one after the other.
- Here is an example of Promise chaining:

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => {
    console.log(data);
    return fetch('https://api.example.com/other-data');
  })
  .then(response => response.json())
  .then(otherData => {
    console.log(otherData);
  })
  .catch(error => {
    console.error(error);
  });
```

In the above example, we first fetch data from one URL, then fetch data from another URL based on the first response, and finally log the second data received.

The catch method would handle any errors that occur during the process. This means you don't need to add error handling to each step, which can greatly simplify your code.

Using async/await to handle promises

Async/await makes writing & reading asynchronous code easier which is built on top of Promises.

- **async**: The async keyword is used to define an asynchronous function.
  An async function returns a Promise, which resolves with the value returned by the async function.
- **await**: The await keyword is used inside an async function to pause the execution of the function until the Promise is resolved. It can only be used inside an async function.
- Here is an example of using async/await:

```
async function delayedGreeting(name) {
  console.log("A Messenger entered the chat...");
  await new Promise(resolve => setTimeout(resolve, 2000));
  console.log(`Hello, ${name}!`);
}

delayedGreeting("Alice");
console.log("First Printed Message!");
```

In the above example, the delayedGreeting function is an async function that pauses for 2 seconds before printing the greeting message. The await keyword is used to pause the function execution until the Promise is resolved.

- One of the biggest advantages of async/await is error handling via try/catch blocks. Here's an example:

```
async function fetchData() {
  try {
```

```
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}

fetchData();
```

In the above example, the try block contains the code that might throw an error, and the catch block handles the error if it occurs. This makes error handling more straightforward and readable.

The async attribute

- The async attribute tells the browser to download the script file asynchronously while continuing to parse the HTML document.
- Once the script is downloaded, the HTML parsing is paused, the script is executed, and then HTML parsing resumes.
- You should use async for independent scripts where the order of execution doesn't matter

The defer attribute

- The defer attribute also downloads the script asynchronously, but it defers the execution of the script until after the HTML document has been fully parsed.
- The defer scripts maintain the order of execution as they appear in the HTML document.
- It's important to note that both async and defer attributes are ignored for inline scripts and only work for external script files.
- When both async and defer attributes are present, the async attribute takes precedence.

Geolocation API

- The Geolocation API provides a way for websites to request the user's location.
- The example below demonstrates the API's getCurrentPosition() method which is used to get the user's current location.

```
navigator.geolocation.getCurrentPosition(
 (position) => {
  console.log("Latitude: " + position.coords.latitude);
  console.log("Longitude: " + position.coords.longitude);
 },
 (error) => {
  console.log("Error: " + error.message);
 }
);
```

In this code, we're calling getCurrentPosition and passing it a function which will be called when the position is successfully obtained.

The position object contains a variety of information, but here we have selected latitude and longitude only.

If there is an issue with getting the position, then the error will be logged to the console.

- It is important to respect the user's privacy and only request their location when necessary.