Event Loop, Promises, Async Await

The Event Loop: A Brief Explanation

What is the Event Loop?

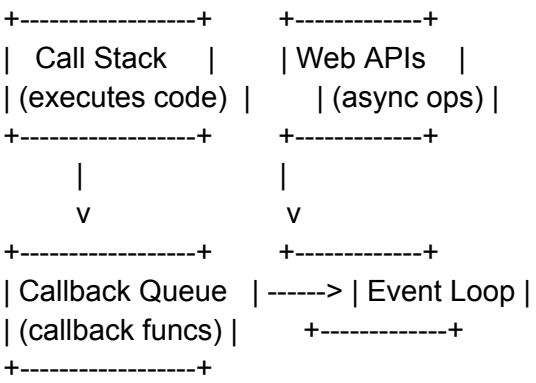
The **event loop** is a fundamental concept in JavaScript that enables asynchronous programming. It allows JavaScript to perform non-blocking operations, even though JavaScript is single-threaded.

How Does the Event Loop Work?

1. **Call Stack**: This is where your function calls are placed and executed. JavaScript is single-threaded, meaning it can do one thing at a time. The call stack handles synchronous code.
2. **Web APIs**: These are provided by the browser (or Node.js environment) and include things like `setTimeout`, `fetch`, DOM events, etc. When asynchronous functions are called, they are sent to the Web APIs.
3. **Callback Queue**: Once an asynchronous operation completes (e.g., a timer finishes, an HTTP request returns), its callback function is placed in the callback queue (also known as the message queue or task queue).
4. **Event Loop**: The event loop continuously checks the call stack and the callback queue. If the call stack is empty, it takes the first callback from the callback queue and pushes it onto the call stack for execution.

Visualizing the Event Loop

plaintextCopy code



Event Loop, Promises, Async Await

Example

Let's see an example to understand how the event loop works:

javascriptCopy code
```
console.log("Start");

setTimeout(() => {
  console.log("Timeout finished");
}, 1000);

console.log("End");
```

Execution Flow:

1. **Call Stack**: The `console.log("Start")` is executed and "Start" is printed.
2. **Web API**: The `setTimeout` function is called, and its callback is sent to the Web APIs with a timer.
3. **Call Stack**: The `console.log("End")` is executed and "End" is printed.
4. **Web API**: After 1000ms, the timer completes, and the callback function is moved to the callback queue.
5. **Event Loop**: The event loop checks the call stack (which is now empty) and moves the callback function from the callback queue to the call stack.
6. **Call Stack**: The callback function `console.log("Timeout finished")` is executed and "Timeout finished" is printed.

Key Takeaways

- **Single-threaded**: JavaScript executes code in a single-threaded manner, using the call stack for synchronous operations.

Event Loop, Promises, Async Await

- **Asynchronous operations**: Managed by Web APIs, which offload tasks that take time (e.g., timers, HTTP requests).
- **Event loop**: Coordinates between the call stack and the callback queue, ensuring asynchronous callbacks are executed when the call stack is clear.

Understanding the event loop is crucial for writing efficient and non-blocking JavaScript code, particularly when dealing with asynchronous operations like network requests, timers, and I/O tasks.

==========

JavaScript Promises: An Overview

What is a Promise?

A **Promise** in JavaScript is an object representing the eventual completion (or failure) of an asynchronous operation and its resulting value. It allows you to write asynchronous code in a more synchronous fashion, avoiding the pitfalls of callback hell.

States of a Promise

A Promise can be in one of three states:

1. **Pending**: The initial state, neither fulfilled nor rejected.
2. **Fulfilled**: The operation completed successfully.
3. **Rejected**: The operation failed.

Creating a Promise

A Promise is created using the `Promise` constructor, which takes a single function (the executor) as an argument. The executor function takes two arguments: `resolve` and `reject`.

javascriptCopy code

```
let promise = new Promise((resolve, reject) => {
  // asynchronous operation
  if (/* operation successful */) {
    resolve("Success!");
  } else {
    reject("Failure!");
  }
});
```

Event Loop, Promises, Async Await

Using Promises

Promises are primarily used with `.then()`, `.catch()`, and `.finally()` methods.

- **.then()**: Executes when the promise is fulfilled.
- **.catch()**: Executes when the promise is rejected.
- **.finally()**: Executes regardless of the promise's outcome (fulfilled or rejected).

javascriptCopy code
promise

```
  .then((result) => {
    console.log(result); // "Success!"
  })
  .catch((error) => {
    console.error(error); // "Failure!"
  })
  .finally(() => {
    console.log("Operation completed");
  });
```

Example: Fetching Data

Here's an example of using a Promise to fetch data from an API:

javascriptCopy code

```
function fetchData(url) {
  return new Promise((resolve, reject) => {

    fetch(url)
      .then(response => {
        if (response.ok) {
          return response.json();
        } else {
          reject("Error: " + response.statusText);
        }
      })
      .then(data => resolve(data))
      .catch(error => reject(error));
  });
}

fetchData("<https://jsonplaceholder.typicode.com/todos> ")
```

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

Chaining Promises

You can chain multiple `.then()` calls to handle a sequence of asynchronous operations.

javascriptCopy code

```
fetchData("<https://api.example.com/data>")
  .then(data => {
    console.log("First fetch:", data);
    return fetchData("<https://api.example.com/other-data>");
  })
  .then(otherData => {
    console.log("Second fetch:", otherData);
  })
  .catch(error => {
    console.error(error);
  });
```

==========

JavaScript Async/Await: An Overview

What is Async/Await?

`async` and `await` are keywords in JavaScript that simplify working with Promises, making asynchronous code look and behave more like synchronous code. They provide a more readable and concise way to handle asynchronous operations.

`async` Functions

An `async` function is a function declared with the `async` keyword, which always returns a Promise. The return value is implicitly wrapped in a Promise if it's not already a Promise.

javascriptCopy code

Event Loop, Promises, Async Await

```
async function myFunction() {
  return "Hello";
}

myFunction().then((result) => console.log(result)); // "Hello"
```

The `await` Keyword

The `await` keyword can only be used inside `async` functions. It pauses the execution of the function, waiting for the Promise to resolve or reject.

```
javascriptCopy code
async function fetchData() {
  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("Data fetched!"), 2000);
  });

  let result = await promise; // wait until the promise resolves
  console.log(result); // "Data fetched!"
}

fetchData();
```

Example: Fetching Data

Here's an example of using `async` and `await` to fetch data from an API:

```
javascriptCopy code
async function fetchData(url) {
  try {
    let response = await fetch(url);
    if (!response.ok) {
      throw new Error("HTTP error " + response.status);
    }
    let data = await response.json();
    return data;
  } catch (error) {
    console.error(error);
  }
}
```

Event Loop, Promises, Async Await

```
fetchData("<https://api.example.com/data>")
  .then(data => {
    console.log(data);
  });
```

Sequential Execution

With `async`/`await`, you can easily handle sequential asynchronous operations without chaining `.then()` calls.

javascriptCopy code

```
async function fetchSequentialData() {
  let firstResponse = await fetch("<https://api.example.com/first-data>");
  let firstData = await firstResponse.json();
  console.log("First data:", firstData);

  let secondResponse = await fetch("<https://api.example.com/second-data>");
  let secondData = await secondResponse.json();
  console.log("Second data:", secondData);
}

fetchSequentialData();
```

Parallel Execution

To run multiple asynchronous operations in parallel, you can use `Promise.all` with `async`/`await`.

javascriptCopy code

```
async function fetchParallelData() {
  let [firstResponse, secondResponse] = await Promise.all([
    fetch("<https://api.example.com/first-data>"),
    fetch("<https://api.example.com/second-data>")
  ]);

  let firstData = await firstResponse.json();
  let secondData = await secondResponse.json();

  console.log("First data:", firstData);
```

```
  console.log("Second data:", secondData);
}

fetchParallelData();
```

Error Handling

Error handling with `async`/`await` is more straightforward using `try...catch`.

```
javascriptCopy code
async function fetchData(url) {
  try {
    let response = await fetch(url);
    if (!response.ok) {
      throw new Error("HTTP error " + response.status);
    }
    let data = await response.json();
    return data;
  } catch (error) {
    console.error("Fetch error:", error);
  }
}

fetchData("<https://api.example.com/data>")
  .then(data => {
    console.log(data);
  });
```

Example: Timeout with Async/Await

Here's an example of using `async`/`await` with a timeout function.

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

async function fetchDataWithTimeout() {
  await timeout(2000);
  console.log("Timeout completed");
```

```
}

fetchDataWithTimeout();
```

Key Takeaways

- **`async`/`await`** provides a more readable and concise way to work with Promises.
- **`async` functions** always return a Promise.
- **`await` pauses** the execution of an `async` function until the Promise is resolved or rejected.
- **Error handling** is simplified with `try...catch`.

By mastering `async`/`await`, students can write asynchronous code that is easier to read and maintain, which is especially important when dealing with complex asynchronous workflows and API interactions.