**Synchronous**

* **Definition**: Code execution that occurs in a sequence, where each operation must complete before the next begins. It follows a blocking model, meaning the program waits for an operation to complete before moving on.
* **Behavior**: When a synchronous operation is performed, the JavaScript runtime stops everything else and waits for that operation to finish. This can lead to delays or "blocking" behavior, especially in scenarios where there are I/O operations or complex computations.
* **Examples**:
  + Traditional loops and straight-line code execution.
  + Certain methods, like **alert()**, are synchronous and block code execution until the user interacts with them.
  + Direct calculations or function calls without asynchronous patterns.

Eg. console.log('First');

console.log('Second');

console.log('Third');

**Asynchronous**

* **Definition**: Code execution that allows tasks to occur without waiting for others to complete. This model doesn't block execution and often involves events, callbacks, promises, or asynchronous functions.
* **Behavior**: Asynchronous code allows the JavaScript runtime to perform other tasks while waiting for a certain operation to complete. This makes it well-suited for I/O operations (like fetching data over a network) where waiting is undesirable.
* **Examples**:
  + **Callbacks**: Functions passed as arguments to other functions to be called later. A typical example is event-driven programming with event listeners.
  + **Promises**: A more structured approach to handling asynchronous operations. Promises represent a value that will be available at some point in the future. They support chaining with **then()** and error handling with **catch()**.
  + **Async/Await**: Syntax introduced in ES2017 (ES8) that allows for a more synchronous-looking way of writing asynchronous code, while still allowing the JavaScript runtime to perform other tasks in the background.
  + **Event Loops and Callbacks**: JavaScript uses an event-driven architecture, where an event loop handles asynchronous tasks. Operations like **setTimeout**, **fetch**, and others trigger the event loop to schedule tasks for later execution, allowing code to continue running in the meantime.

**Key Differences**

* **Execution Flow**: Synchronous code has a linear, sequential execution flow. Asynchronous code allows for non-blocking execution, where tasks can be scheduled for later completion.
* **Performance and Responsiveness**: Synchronous code can lead to performance bottlenecks and unresponsive behavior if long-running operations are blocking. Asynchronous code, by contrast, is more conducive to responsive and scalable applications.
* **Use Cases**: Use synchronous patterns for simple computations and scenarios where blocking is acceptable. Use asynchronous patterns for network operations, file I/O, database queries, or any operation that could take time and shouldn't block the main execution flow.

console.log('First');

setTimeout(() => {

console.log('Second');

}, 0);

console.log('Third');

**Promises**

**What is a Promise?**

A Promise is an object representing an eventual completion or failure of an asynchronous operation. It has three states:

1. **Pending**: The initial state, where the promise has not yet resolved or rejected.
2. **Fulfilled**: The promise successfully completed, and a result is available.
3. **Rejected**: The promise encountered an error, and an error message or exception is available.

**Creating a Promise**

To create a Promise, you use the **Promise** constructor and provide a function with two arguments: **resolve** and **reject**. The function body contains the asynchronous operation, and when it's complete, you call either **resolve** (for success) or **reject** (for failure).

const myPromise = new Promise((resolve, reject)

=> { const success = true; // Simulate a condition

if (success) { resolve("Operation was successful!"); }

else { reject("Something went wrong."); } });

**Using Promises with then and catch**

To handle the outcome of a Promise, you use the **then()** method to define what should happen if the promise is fulfilled, and the **catch()** method for when it is rejected.

myPromise .then((result) => { console.log(result); // Output: "Operation was successful!" })

.catch((error) => { console.error(error); // Will not execute in this example });

**Chaining Promises**

One of the significant advantages of Promises is that they can be chained, allowing you to create a sequence of asynchronous operations.

**Making a Simple GET Request**

fetch("https://jsonplaceholder.typicode.com/posts/1")

.then((response) => {

if (!response.ok) {

throw new Error("Network response was not ok");

}

return response.json(); // Convert the response to JSON

})

.then((data) => {

console.log(data); // Output: JSON data from the API

})

.catch((error) => {

console.error("Fetch error:", error);

});

**async/await** is a syntactic feature in JavaScript designed to handle asynchronous code in a way that looks and feels like synchronous code. Introduced in ECMAScript 2017 (ES8), **async/await** allows you to write asynchronous code in a more readable and concise manner, making it easier to understand the flow of execution.

**What is async/await?**

* **async Keyword**: This keyword is used to declare an asynchronous function. An asynchronous function automatically returns a Promise, even if it doesn't contain an explicit **return** statement.
* **await Keyword**: Used inside an **async** function to pause execution until the Promise being awaited is resolved (or rejected). This creates a sense of synchronous flow, although the underlying execution remains asynchronous and non-blocking.

**Basic Example**

async function fetchData() {

const response = await fetch("https://jsonplaceholder.typicode.com/posts/1");

const data = await response.json();

return data;

}

fetchData().then((result) => {

console.log(result);

});

In this example:

* The **fetchData** function is marked as **async**.
* Inside **fetchData**, the **await** keyword is used to wait for the **fetch** call to complete and the **response.json()** to resolve.
* The function ultimately returns a Promise, which can be handled with **then**.

**Handling Errors with async/await**

To handle errors, you can use a **try/catch** block inside an **async** function. If an error is thrown, it can be caught and handled within the **catch** block.

async function fetchData() {

try {

const response = await fetch("https://jsonplaceholder.typicode.com/posts/1");

if (!response.ok) {

throw new Error(`HTTP error! status: ${response.status}`);

}

const data = await response.json();

return data;

} catch (error) {

console.error("Error fetching data:", error);

}

}

fetchData();