Synchronous and Asynchronous

Synchronous: In synchronous operations, code is executed sequentially, one line at a time. Each line must wait for the previous one to finish executing before it can start. This can sometimes lead to blocking behavior, where one task prevents another from executing until it's complete.

```
console.log("Start");
console.log("Middle");
console.log("End");
// start
// middle
// end
```

In this synchronous code snippet, "Start" will be logged first, followed by "Middle", and then "End".

Asynchronous: Asynchronous operations allow code to execute independently from the main program flow. This means that while one operation is being processed, the program can continue to execute other tasks. Asynchronous operations are typically used for tasks that may take some time to complete, such as fetching data from a server or reading a file. In JavaScript, common asynchronous operations are handled using callbacks, promises, or async/await syntax.

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

setTimeout(()=>{
  console.log("middle")
}),2000;
```

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


// start
// end
// middle
```

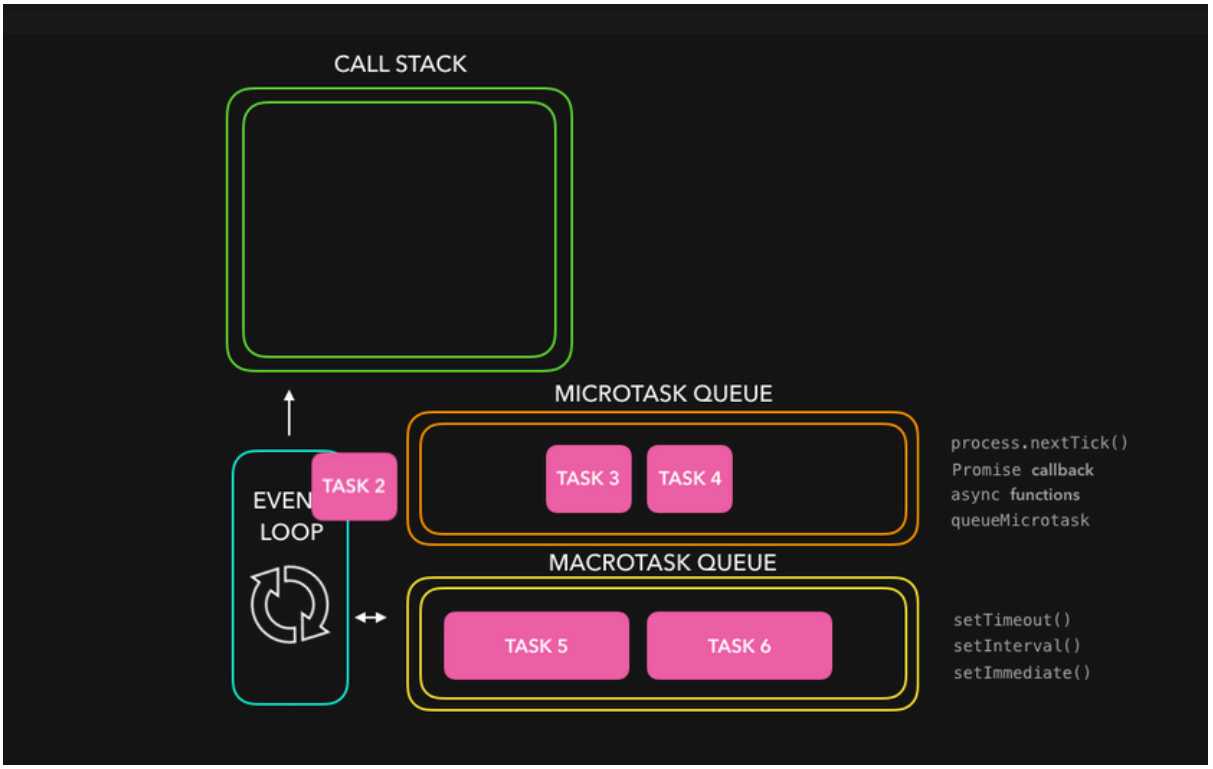
In this example, "Start" is logged first, then after a delay of 2000 milliseconds, "End" is logged, followed by " Middle ".

How JS works

1. **Event Loop**: The event loop is a mechanism in JavaScript that continuously checks the call stack and the callback queue. It ensures that the execution of code is done in the right order, especially when dealing with asynchronous operations.
2. **Call Stack:** The call stack is a data structure that keeps track of function calls in the code. Whenever a function is called, it's added to the top of the call stack. When a function completes, it's removed from the stack.
3. **Callback que:**

   "callback queue," also known as the "task queue," is a part of JavaScript's event loop mechanism that manages asynchronous tasks or callbacks for execution. When an asynchronous operation such as a timer (setTimeout or setInterval) or an event (such as user interaction or network response) completes, its associated callback function is placed in the callback queue.

4. **Micro task que**
   Microtask queue in the event loop handles small units of asynchronous work with higher priority, ensuring they are executed before regular tasks. It's commonly used for tasks associated with promises



5. **CallStack Overflow:** If the call stack grows too large, typically due to infinite recursion or excessive nested function calls, it can exceed the available memory allocated for the stack. This results in a "stack overflow" error and crashes the program.

Callbacks

callback is a function that is passed as an argument to another function and is executed after some operation has been completed. This is a powerful feature that allows for asynchronous programming, enabling tasks to run concurrently without blocking the main execution thread.

Defining the Callback:

```
// Step 1: Define the callback function
function myCallback() {
  console.log("Callback function executed!");
```

```
}
```

The callback function can be defined separately or inline.

```
// Defining separately
function myCallback() {
  console.log("Callback executed!");
}
doSomething(myCallback);

// Defining inline
doSomething(function () {
  console.log("Callback executed!");
});
```

Execute the Callback

The outer function calls the callback function at the appropriate time.

```
    // Step 2: Define a function that takes another function as a parameter
function doSomething(callback) {
  console.log("Doing something...");

  // Step 3: Execute the callback function
  callback();
}
```

Pass the Callback Function as a argument:

```
// Step 4: Pass the callback function as an argument
doSomething(myCallback);
```

Ex-1

```
function myFirst() {
  console.log("Hello");
}

function mySecond(a) {
  console.log("Goodbye");
}

var a=myFirst();
mySecond(a);
//Hello

//Goodbye
```

Ex-2

```
function myFirst() {
  console.log("Hello");
}

function mySecond() {
   myFirst()
```

```
  console.log("Goodbye");
}

mySecond();
```

The problem with the first example above, is that you have to call two functions to display the result.

The problem with the second example, is that you cannot prevent the function from displaying the result when it invokes every time.

```
function myFirst() {
  console.log("Hello");
}

function mySecond(a) {
  console.log("Goodbye");
}

mySecond(myFirst());//can call two functions at a time
mySecond()//one function
```

Usage of Callbacks

Callbacks are commonly used in situations where you want to perform tasks asynchronously, such as:

- Event Handling
- Asynchronous Operations
- Higher-Order Functions
- Array Methods

Array for each

Array.forEach() is a method in JavaScript used to iterate over elements in an array. It executes a provided function once for each array element

array.forEach(function(value, index, array) {

    // Your code here

});

value: The current item in a array.

index (optional): The index of the item in the array.

array (optional): The array that forEach()

```
const numbers = [1, 2, 3, 4, 5];

numbers.forEach(function(val, index) {
    console.log(`Element at index ${index} is ${val }`);
});
//output
// Element at index 0 is 1
// Element at index 1 is 2
// Element at index 2 is 3
// Element at index 3 is 4
// Element at index 4 is 5
```

You can also use arrow functions for a more concise syntax:

```
numbers.forEach((number, index) => {
    console.log(`Element at index ${index} is ${number}`);
});
```

One important thing to note about forEach() is that it **doesn't return anything**. It simply **iterates over the array**. If you need to transform the elements of the array and create a new array based on those transformations, you might want to use methods like map() instead.

```
array.forEach(function(element) {
    return element * 2; // This return statement has no effect
});
```

Map method

The map() method in JavaScript is used to create a new array by calling a provided function on every element in the calling array. It doesn't change the original array; instead, it returns a new array with the results of applying the provided function to each element.

```
const newArray = array.map(function callback(currentValue, index, array) {
  // Return element for newArray
});
```

value: The current item in a array.

index (optional): The index of the item in the array.

array (optional): The array that forEach()

```
const numbers = [1, 2, 3, 4, 5];

const doubledNumbers = numbers.map(function(number) {
  return number * 2; // Return value determines the value
});
```

```
console.log(doubledNumbers); // Output: [2, 4, 6, 8, 10]
```

You can also use arrow functions for more concise syntax:

```
const numbers = [1, 2, 3, 4, 5];

const doubledNumbers = numbers.map(number => number * 2);

console.log(doubledNumbers); // Output: [2, 4, 6, 8, 10]
```

Filter method

The filter() method in JavaScript is used to create a new array with all elements that pass a certain condition. It doesn't change the original array; instead, it returns a new array containing only the elements for which the provided filtering function returns true.

```
const numbers = [1, 2, 3, 4, 5];

const filteredNumbers = numbers.filter(function(number) {
  return number >3;
});

console.log(filteredNumbers); // Output: [4, 5]
```

You can also use arrow functions for more concise syntax:

```
const numbers = [1, 2, 3, 4, 5];
```

```
const evenNumbers = numbers.filter(number => number % 2 === 0);

console.log(evenNumbers); // Output: [2, 4]
```

Use **map method** when you need to transform each element of an array and create a new array with the transformed values. when you want to perform operatons

Use **filter method** when you need to filter elements from an array based on some criteria and create a new array with only the elements that meet that criteria.when you want to perform filteration.

Use **for each method** When you need to perform an action for each element of an array without necessarily creating a new array or transforming the elements. When you want to perform itterations.

Reduce method

The **reduce()** method in JavaScript is used to reduce the elements of an array to a single value.

Accumulator (acc): The accumulated value computed from previous iterations.

Current Value (cur): The current element being processed in the array.

Current Index (ind) (optional): The index of the current element being processed in the array.

Source Array (src) (optional): The array reduce() was called upon.

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce((accumulator, currentValue) => {
  return accumulator + currentValue;
}, 0);

console.log(sum); // Output: 15 (1 + 2 + 3 + 4 + 5)
```

reduce() can be used for various tasks, such as calculating the maximum or minimum value in an array, concatenating elements into a string, or performing more complex operations. It's a powerful method for aggregating data in arrays.

Reduceright method

The reduceRight() method in JavaScript is used to reduce the elements of an array to a single value, but it processes the array from right to left.

- **Accumulator (acc)**: The accumulated value computed from previous iterations.
- **Current Value (cur)**: The current element being processed in the array.
- **Current Index (ind)** (optional): The index of the current element being processed in the array.
- **Source Array (src)** (optional): The array reduceRight() was called upon.

```
const numbers = [1, 2, 3, 4, 5];

const sum = numbers.reduceRight((accumulator, currentValue) => {
return accumulator + currentValue;
}, 0);
```

```
console.log(sum); // Output: 15 (5 + 4 + 3 + 2 + 1)
```

The reduceRight() method can be used for various tasks, such as calculating the maximum or minimum value in an array, concatenating elements into a string, or performing more complex operations.

It's a powerful method for aggregating data in arrays, especially when the order of operations from right to left is significant.

Sort method

The **sort()** method in JavaScript is used to sort the elements of an array in place and returns the sorted array.

```
const numbers = [4, 2, 5, 1, 3];
numbers.sort((a, b) => a - b);
console.log(numbers); // Output: [1, 2, 3, 4, 5]


const fruits = ['banana', 'apple', 'orange', 'grape'];
fruits.sort();
console.log(fruits); // Output: ['apple', 'banana', 'grape', 'orange']
// method will sort the elements alphabetically as strings.
```

In this example, the **compareFunction (a, b) => a - b** sorts the numbers in ascending order. If **a - b** is negative, **a** comes before **b**, if it's positive, **b** comes before **a**, and if it's zero, the order remains unchanged.

The **some()** and **every()** methods in JavaScript are both used to check the elements of an array against certain conditions.

some() Method:

Purpose: It checks if at least one element in the array satisfies the provided testing function. It returns true if any element passes the test; otherwise, it returns false.

Syntax: array.some(callback(element, index, array))

Return Value: true if at least one element passes the test; otherwise, false.

```
const numbers = [1, 2, 3, 4, 5];
const numbers2 = numbers.some(number => number >3);
console.log(number2); // Output: true
```

every() Method:

- Purpose: It checks if all elements in the array satisfy the provided testing function. It returns **true** if all elements pass the test; otherwise, it returns **false**.
- Syntax: **array.every(callback(element, index, array))**
- Return Value: **true** if all elements pass the test; otherwise, **false**.

```
const numbers = [1, 2, 3, 4, 5];
const numbers2 = numbers.every(number => number >3);
console.log(numbers2); // Output: false
```