```
clearTimeout(id);
console.log('setTimeout is stopped.');
```

JavaScript Spread Operator

The JavaScript spread operator ... is used to expand or spread out elements of an iterable, such as an array, string, or object.
This makes it incredibly useful for tasks like combining arrays, passing elements to functions as separate arguments, or even copying arrays.

```
let numbers = [1, 2, 3];

// equivalent to
// console.log(numbers[0], numbers[1], numbers[2])
console.log(...numbers);

// Output: 1 2 3
```

JavaScript Spread Operator Inside Arrays

We can also use the spread operator inside arrays to expand the elements of another array. For example,

```
let fruits = ["Apple", "Banana", "Cherry"];

// add fruits array to moreFruits1
// without using the ... operator
let moreFruits1 = ["Dragonfruit", fruits, "Elderberry"];

// spread fruits array within moreFruits2 array
let moreFruits2 = ["Dragonfruit", ...fruits, "Elderberry"];

console.log(moreFruits1);
console.log(moreFruits2);
```

Run Code

Output

```
[ 'Dragonfruit', [ 'Apple', 'Banana', 'Cherry' ], 'Elderberry' ]
[ 'Dragonfruit', 'Apple', 'Banana', 'Cherry', 'Elderberry' ]
```

Here, ...fruits expands the fruits array inside the moreFruits2 array, which results in moreFruits2 consisting only of individual string elements and no inner arrays.
On the other hand, the moreFruits1 array consists of an inner array because we didn't expand the fruits array inside it.

Clone Array Using Spread Operator

In JavaScript, objects are assigned by reference and not by values. For example,

```
let arr1 = [ 1, 2, 3];
let arr2 = arr1;

console.log(arr1); // [1, 2, 3]
console.log(arr2); // [1, 2, 3]

// append an item to arr1
arr1.push(4);

console.log(arr1); // [1, 2, 3, 4]
console.log(arr2); // [1, 2, 3, 4]
```

Here, both variables `arr1` and `arr2` are referring to the same array. Hence, a change in one variable results in a change in both variables.

However, if you want to copy arrays so that they do not refer to the same array, you can use the spread operator. This way, the change in one array is not reflected in the other. For example,

```
let arr1 = [ 1, 2, 3];

// copy using spread syntax
let arr2 = [...arr1];

console.log(arr1); // [1, 2, 3]
console.log(arr2); // [1, 2, 3]

// append an item to arr1
arr1.push(4);

console.log(arr1); // [1, 2, 3, 4]
console.log(arr2); // [1, 2, 3]
```

Spread Operator With Object

You can also use the spread operator with object literals. For example,

```
let obj1 = { x : 1, y : 2 };
let obj2 = { z : 3 };

// use the spread operator to add
// members of obj1 and obj2 to obj3
let obj3 = {...obj1, ...obj2};

// add obj1 and obj2 without spread operator
let obj4 = {obj1, obj2};

console.log("obj3 =", obj3);
console.log("obj4 =", obj4);
```

```
obj3 = { x: 1, y: 2, z: 3 }
obj4 = { obj1: { x: 1, y: 2 }, obj2: { z: 3 } }
```

Here, the properties of obj1 and obj2 are added to obj3 using the spread operator.
However, when we add those two objects to obj4 without using the spread operator, we get obj1 and obj2 as keys for obj4.

JavaScript Rest Parameter

When the spread operator is used as a parameter, it is known as the rest parameter.
You can accept multiple arguments in a function call using the rest parameter. For example,

```
let printArray = function(...args) {
    console.log(args);
}

// pass a single argument
printArray(3);

// pass multiple arguments
printArray(4, 5, 6);
```

Run Code
Output

```
[ 3 ]
[ 4, 5, 6 ]
```

Here,

- When a single argument is passed to printArray(), the rest parameter takes only one parameter.
- When three arguments are passed, the rest parameter takes all three parameters.

Note: Using the rest parameter will pass the arguments as array elements.

CallBack Function

```
// function
function greet(name) {
    console.log('Hi' + ' ' + name);
}

greet('Peter'); // Hi Peter
```

In JavaScript, you can also pass a function as an argument to a function. This function that is passed as an argument inside of another function is called a **callback function**. For example,

```
// function
function greet(name, callback) {
    console.log('Hi' + ' ' + name);
    callback();
}

// callback function
function callMe() {
    console.log('I am callback function');
}

// passing function as an argument
greet('Peter', callMe);
```

Benefit of Callback Function

The benefit of using a callback function is that you can wait for the result of a previous function call and then execute another function call.

```
//  program that shows the delay in execution
```

```
function greet() {
   console.log('Hello world');
}

function sayName(name) {
   console.log('Hello' + ' ' + name);
}

// calling the function
setTimeout(greet, 2000);
sayName('John');
```

Output

```
Hello John
Hello world
```

JavaScript Promise and Promise Chaining

In JavaScript, a promise is a good way to handle asynchronous operations. It is used to find out if the asynchronous operation is successfully completed or not.
A promise may have one of three states.

- Pending
- Fulfilled
- Rejected

A promise starts in a pending state. That means the process is not complete. If the operation is successful, the process ends in a fulfilled state. And, if an error occurs, the process ends in a rejected state.
For example, when you request data from the server by using a promise, it will be in a pending state. When the data arrives successfully, it will be in a fulfilled state. If an error occurs, then it will be in a rejected state.

Create a Promise

To create a promise object, we use the `Promise()` constructor.

```
let promise = new Promise(function(resolve, reject){
      //do something
});
```

The Promise() constructor takes a function as an argument. The function also accepts two functions resolve() and reject().
If the promise returns successfully, the resolve() function is called. And, if an error occurs, the reject() function is called.

Example - 1

```
var promise = new Promise(function(resolve, reject) {
    setTimeout(function() {
        resolve('hello world');
    }, 2000);
});

promise.then(function(data) {
    console.log(data);
});
```

Example - 2

```
var promise = new Promise(function(resolve, reject) {
   setTimeout(function() {
      resolve('hello world');
   }, 2000);
});

promise.then(function(data) {
   console.log(data + ' 1');
});

promise.then(function(data) {
   console.log(data + ' 2');
});

promise.then(function(data) {
   console.log(data + ' 3');
```

```
});
```

`Example -3`

```
const count = true;

let countValue = new Promise(function (resolve, reject) {
    if (count) {
        resolve("There is a count value.");
    } else {
        reject("There is no count value");
    }
});

console.log(countValue);
```

Output

```
Promise {<resolved>: "There is a count value."}
```

Promise Chaining

Promises are useful when you have to handle more than one asynchronous task, one after another. For that, we use promise chaining.

You can perform an operation after a promise is resolved using methods `then()`, `catch()` and `finally()`.

JavaScript then() method

The `then()` method is used with the callback when the promise is successfully fulfilled or resolved.

Example 2: Chaining the Promise with then()

```
// returns a promise
```

```
let countValue = new Promise(function (resolve, reject) {
  resolve("Promise resolved");
});

// executes when promise is resolved successfully

countValue
  .then(function successValue(result) {
    console.log(result);
  })

  .then(function successValue1() {
    console.log("You can call multiple functions this way.");
  });
```

Output
Promise resolved
You can call multiple functions this way.

In the above program, the then() method is used to chain the functions to the promise. The then() method is called when the promise is resolved successfully.
You can chain multiple then() methods with the promise.

JavaScript catch() method

The `catch()` method is used with the callback when the promise is rejected or if an error occurs. For example,

```
// returns a promise
let countValue = new Promise(function (resolve, reject) {
   reject('Promise rejected');
});

// executes when promise is resolved successfully
countValue.then(
    function successValue(result) {
        console.log(result);
    },
 )

// executes if there is an error
```

```
.catch(
   function errorValue(result) {
      console.log(result);
   }
);
```

Output

```
Promise rejected
```

JavaScript async/await

We use the `async` keyword with a function to represent that the function is an asynchronous function. The async function returns a promise.

The syntax of `async` function is:

```
async function name(parameter1, parameter2, ...paramaterN) {
    // statements
}
```

Here,

- name - name of the function
- parameters - parameters that are passed to the function

Example: Async Function

```
// async function example

async function f() {
    console.log('Async function.');
    return Promise.resolve(1);
}
```

```
f();
```

Output

```
Async function.
```

In the above program, the `async` keyword is used before the function to represent that the function is asynchronous.

Since this function returns a promise, you can use the chaining method `then()` like this:

```
async function f() {
    console.log('Async function.');
    return Promise.resolve(1);
}

f().then(function(result) {
    console.log(result)
});
```

Output

```
Async function
1
```

In the above program, the `f()` function is resolved and the `then()` method gets executed.

JavaScript await Keyword

The `await` keyword is used inside the `async` function to wait for the asynchronous operation.

The syntax to use await is:

```
let result = await promise;
```

The use of `await` pauses the async function until the promise returns a result (resolve or reject) value. For example,

```
// a promise
let promise = new Promise(function (resolve, reject) {
    setTimeout(function () {
    resolve('Promise resolved')}, 4000);
});

// async function
async function asyncFunc() {

    // wait until the promise resolves
    let result = await promise;

    console.log(result);
    console.log('hello');
}

// calling the async function
asyncFunc();
```

Output

```
Promise resolved
hello
```

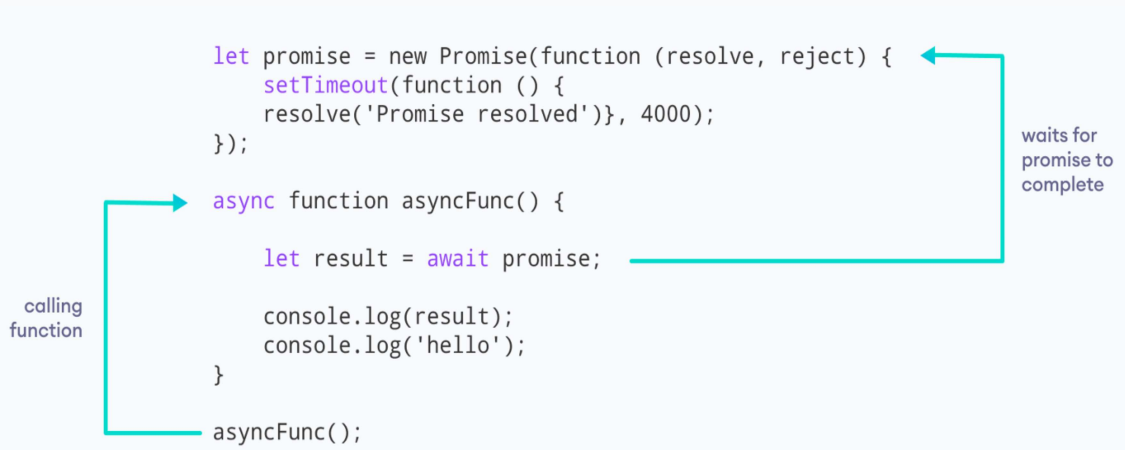
In the above program, a `Promise` object is created and it gets resolved after 4000 milliseconds. Here, the `asyncFunc()` function is written using the `async` function.

The `await` keyword waits for the promise to be complete (resolve or reject).

```
let result = await promise;
```

Hence, `hello` is displayed only after promise value is available to the `result` variable.

In the above program, if `await` is not used, `hello` is displayed before `Promise resolved`.



Working of async/await function

> **Note: You can use `await` only inside of async functions.**

The async function allows the asynchronous method to be executed in a seemingly synchronous way. Though the operation is asynchronous, it seems that the operation is executed in synchronous manner.

This can be useful if there are multiple promises in the program. For example,

```
let promise1;
let promise2;
let promise3;

async function asyncFunc() {
    let result1 = await promise1;
    let result2 = await promise2;
```

```
    let result3 = await promise3;

    console.log(result1);
    console.log(result1);
    console.log(result1);
}
```

In the above program, `await` waits for each promise to be complete.

Error Handling

While using the `async` function, you write the code in a synchronous manner. And you can also use the `catch()` method to catch the error. For example,

```
asyncFunc().catch(
    // catch error and do something
)
```

The other way you can handle an error is by using `try/catch` block. For example,

```
// a promise
let promise = new Promise(function (resolve, reject) {
    setTimeout(function () {
    resolve('Promise resolved')}, 4000);
});

// async function
async function asyncFunc() {
    try {
        // wait until the promise resolves
        let result = await promise;

        console.log(result);
    }
    catch(error) {
```

```
        console.log(error);
    }
}

// calling the async function
asyncFunc(); // Promise resolved
```

In the above program, we have used `try/catch` block to handle the errors. If the program runs successfully, it will go to the `try` block. And if the program throws an error, it will go to the `catch` block.

To learn more about `try/catch` in detail, visit JavaScript JavaScript try/catch.

Benefits of Using async Function

- **The code is more readable than using a callback or a promise.**
- **Error handling is simpler.**
- **Debugging is easier.**

> Note: These two keywords `async/await` were introduced in the newer version of JavaScript (ES8). Some older browsers may not support the use of async/await. To learn more, visit JavaScript async/await browser support.