## Asynchronous vs. synchronous programming

Ultimately, the choice comes down to operational dependencies. Do you want the start of an operation to be dependent on the completion of another operation? Or do you want it to run independently?
**Asynchronous** is a non-blocking architecture, so the execution of one task isn't dependent on another. Tasks can run simultaneously.
**Synchronous** is a blocking architecture, so the execution of each operation is dependent on the completion of the one before it. Each task requires an answer before moving on to the next iteration.
The differences between asynchronous and synchronous include:

- **Async** is multi-thread, which means operations or programs can run in parallel.
- **Sync** is single-thread, so only one operation or program will run at a time.
- **Async** is non-blocking, which means it will send multiple requests to a server.
- **Sync** is blocking — it will only send the server one request at a time and will wait for that request to be answered by the server.
- **Async** increases throughput because multiple operations can run at the same time.
- **Sync** is slower and more methodical.

## Javascript setTimeout()

The setTimeout() method executes a block of code after the specified time. The method executes the code only once.

The commonly used syntax of JavaScript setTimeout is:

setTimeout(function, milliseconds);

Its parameters are:

- **function** - a function containing a block of code
- **milliseconds** - the time after which the function is executed

The setTimeout() method returns an **intervalID**, which is a positive integer.

**Example 1: Display a Text Once After 3 Second**

```
// program to display a text using setTimeout method

function greet() {

    console.log('Hello world');

}

setTimeout(greet, 3000);

console.log('This message is shown first’);
```

Output:

```
This message is shown first
Hello world
```

In the above program, the setTimeout() method calls the greet() function after **3000** milliseconds (**3** second).

Hence, the program displays the text Hello world only once after **3** seconds.

```
// program to display a text using setTimeout method
function greet() {
    console.log('Hello world');
}

let intervalId = setTimeout(greet, 3000);
console.log('Id: ' + intervalId);
```

Output:

```
Id: 3
Hello world
```

**Example 2: Display Time Every 3 Second**

```
// program to display time every 3 seconds
function showTime() {

    // return new date and time
    let dateTime= new Date();

    // returns the current local time
    let time = dateTime.toLocaleTimeString();

    console.log(time)

    // display the time after 3 seconds
     setTimeout(showTime, 3000);
}

// calling the function
showTime();
```

The above program displays the time every **3** seconds.

The setTimeout() method calls the function only once after the time interval (here **3** seconds).

However, in the above program, since the function is calling itself, the program displays the time every **3** seconds.

### JavaScript clearTimeout()

As you have seen in the above example, the program executes a block of code after the specified time interval. If you want to stop this function call, you can use the clearTimeout() method.

### Example 3: Use clearTimeout() Method

```
// program to stop the setTimeout() method

let count = 0;

// function creation
```

```
function increaseCount(){

    // increasing the count by 1
    count += 1;
    console.log(count)
}

let id = setTimeout(increaseCount, 3000);

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

Output:
setTimeout is stopped.

In the above program, the setTimeout() method is used to increase the value of count after 3 seconds. However, the clearTimeout() method stops the function call of the setTimeout() method. Hence, the count value is not increased.

```
// program to display a name
function greet(name, lastName) {
    console.log('Hello' + ' ' + name + ' ' + lastName);
}

// passing argument to setTimeout
setTimeout(greet, 1000, 'John', 'Doe');
```

Output:
Hello John Doe

In the above program, two parameters John and Doe are passed to the setTimeout() method. These two parameters are the arguments that will be passed to the function (here, greet() function) that is defined inside the setTimeout() method.

## JavaScript CallBack Function

A function is a block of code that performs a certain task when called. For example,

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

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

In the above program, a string value is passed as an argument to the greet() function.

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);
```

Output:
Hi Peter
I am callback function

In the above program, there are two functions. While calling the greet() function, two arguments (a string value and a function) are passed.

The callMe() function is a callback function.

### 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.

In this example, we are going to use the setTimeout() method to mimic the program that takes time to execute, such as data coming from the server.

### Example: Program with setTimeout()

```
//  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

Here, the greet() function is called after **2000** milliseconds (**2** seconds). During this wait, the sayName('John'); is executed. That is why Hello John is printed before Hello world.

The above code is executed asynchronously (the second function; sayName() does not wait for the first function; greet() to complete).

### Example: Using a Callback Function

In the above example, the second function does not wait for the first function to be complete. However, if you want to wait for the result of the previous function call before the next statement is executed, you can use a callback function. For example,

```
// Callback Function Example
function greet(name, myFunction) {
    console.log('Hello world');

    // callback function
    // executed only after the greet() is executed
    myFunction(name);
}

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

// calling the function after 2 seconds
setTimeout(greet, 2000, 'John', sayName);
```

Output:
Hello world
Hello John

### 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: Program with a Promise**

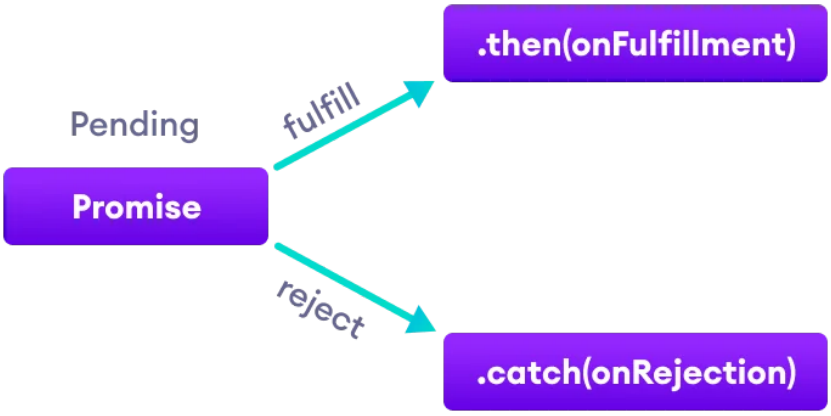
```
const count = true;

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.”}

In the above program, a Promise object is created that takes two functions: resolve() and reject(). resolve() is used if the process is successful and reject() is used when an error occurs in the promise.

The promise is resolved if the value of count is true.




**JavaScript 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().

**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.

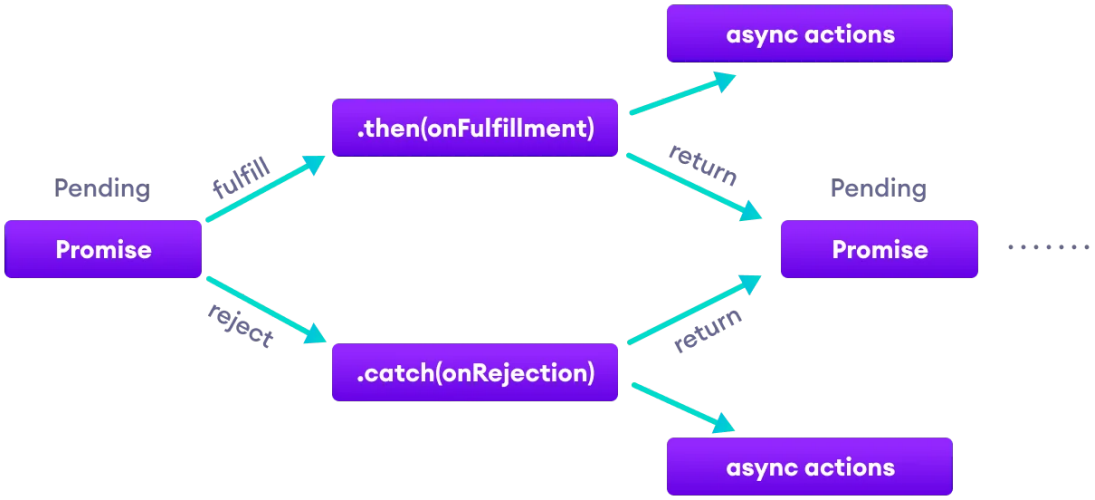
**JavaScript catch() method**

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 Promise Versus Callback

Promises are similar to callback functions in a sense that they both can be used to handle asynchronous tasks.

JavaScript callback functions can also be used to perform synchronous tasks.

Their differences can be summarized in the following points:

## JavaScript Promise

1. The syntax is user-friendly and easy to read.
2. Error handling is easier to manage.

## JavaScript finally() method

You can also use the finally() method with promises. The finally() method gets executed when the promise is either resolved successfully or rejected. For example,

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

// add other blocks of code
countValue.finally(
    function greet() {
        console.log('This code is executed.');
    }
);
```

Output:
  This code is executed

## Javscript 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.

```
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,

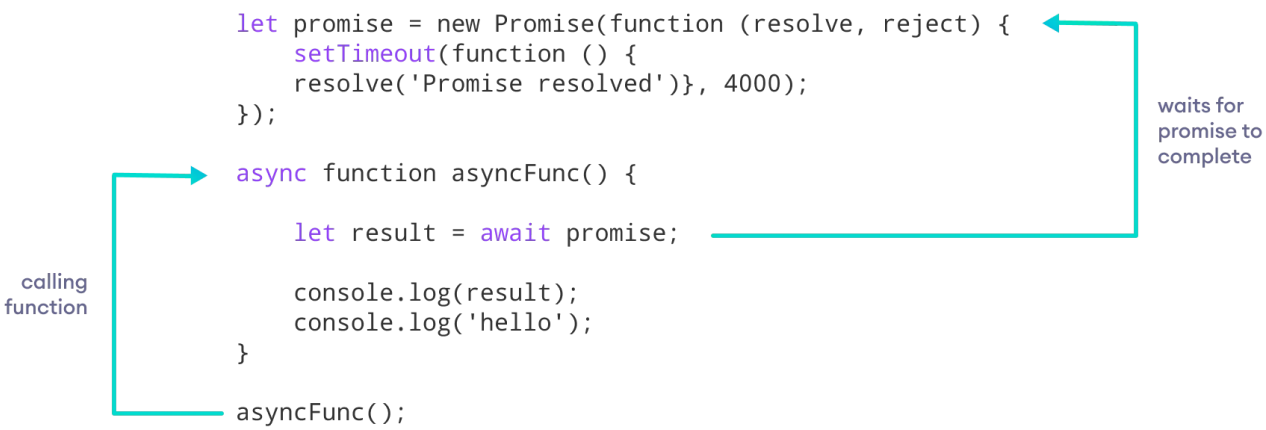
```
// a promise
let promise = new Promise(function (resolve, reject) {
    setTimeout(function () {
    resolve('Promise resolved')}, 4000);
});

// 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 rejected
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).

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.

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,

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
```

**Javascript setInterval()**

In JavaScript, a block of code can be executed in specified time intervals. These time intervals are called timing events.

There are two methods for executing code at specific intervals. They are:

- setInterval()
- setTimeout()

**JavaScript setInterval()**

The setInterval() method repeats a block of code at every given timing event.

Its parameters are:

- **function** - a function containing a block of code
- **milliseconds** - the time interval between the execution of the function

The setInterval() method returns an **intervalID** which is a positive integer.

### Example 1: Display a Text Once Every 1 Second

```
// program to display a text using setInterval method
function greet() {
    console.log('Hello world');
}

setInterval(greet, 1000);
```

### Output

```
Hello world
Hello world
Hello world
Hello world
```

In the above program, the setInterval() method calls the greet() function every **1000** milliseconds(**1** second).

### Example 2: Display Time Every 5 Seconds

```
// program to display time every 5 seconds
function showTime() {

    // return new date and time
```

```
    let dateTime= new Date();

    // return the time
    let time = dateTime.toLocaleTimeString();

    console.log(time)
}

let display = setInterval(showTime, 5000);
```

Output:
"5:15:28 PM"
"5:15:33 PM"
"5:15:38 PM"
….

The above program displays the current time once every **5** seconds.

new Date() gives the current date and time. And toLocaleTimeString() returns the current time

### JavaScript clearInterval()

As you have seen in the above example, the program executes a block of code at every specified time interval. If you want to stop this function call, then you can use the clearInterval() method.

### Example 3: Use clearInterval() Method

```
// program to stop the setInterval() method after five times

let count = 0;

// function creation
let interval = setInterval(function(){

    // increasing the count by 1
    count += 1;
```

```
    // when count equals to 5, stop the function
    if(count === 5){
        clearInterval(interval);
    }

    // display the current time
    let dateTime= new Date();
    let time = dateTime.toLocaleTimeString();
    console.log(time);

}, 2000);
```

Output:
4:47:41 PM
4:47:43 PM
4:47:45 PM
4:47:47 PM
4:47:49 PM

In the above program, the setInterval() method is used to display the current time every **2** seconds. The clearInterval() method stops the function call after **5** times.

**JavaScript and JSON**

JSON stands for Javascript Object Notation. JSON is a text-based data format that is used to store and transfer data. For example,

```
// JSON syntax
{
    "name": "John",
    "age": 22,
    "gender": "male",

}
```

In JSON, the data are in **key/value** pairs separated by a comma ,.

JSON was derived from JavaScript. So, the JSON syntax resembles JavaScript object literal syntax. However, the JSON format can be accessed and be created by other programming languages too.

### JSON Data

JSON data consists of **key/value** pairs similar to JavaScript object properties. The key and values are written in double quotes separated by a colon :. For example,

### JSON Object

The JSON object is written inside curly braces { }. JSON objects can contain multiple **key/value** pairs. For example,

```
// JSON object
{ "name": "John", "age": 22 }
```

### JSON Array

JSON array is written inside square brackets [ ]. For example,

```
// JSON array
[ "apple", "mango", "banana"]

// JSON array containing objects
[
   { "name": "John", "age": 22 },
   { "name": "Peter", "age": 20 }.
   { "name": "Mark", "age": 23 }
]
```

### Accessing JSON Data

You can access JSON data using the dot notation. For example

```
// JSON object
const data = {
   "name": "John",
```

```
    "age": 22,
    "hobby": {
        "reading" : true,
        "gaming" : false,
        "sport" : "football"
    },
    "class" : ["JavaScript", "HTML", "CSS"]
}

// accessing JSON object
console.log(data.name); // John
console.log(data.hobby); // { gaming: false, reading: true, sport:
"football"}

console.log(data.hobby.sport); // football
console.log(data.class[1]); // HTML
```

You can also use square bracket syntax [] to access JSON data. For example,

```
// JSON object
const data = {
    "name": "John",
    "age": 22
}

// accessing JSON object
console.log(data["name"]); // John
```

| JSON | JavaScript Objects |
|---|---|
| The key in key/value pair should be in double quotes. | The key in key/value pair can be without double quotes. |
| JSON cannot contain functions. | JavaScript objects can contain functions. |
| JSON can be created and used by other programming languages. | JavaScript objects can only be used in JavaScript. |

## Converting JSON to JavaScript Object

You can convert JSON data to a JavaScript object using the built-in JSON.parse() function. For example,

```
// json object
const jsonData = '{ "name": "John", "age": 22 }';

// converting to JavaScript object
const obj = JSON.parse(jsonData);

// accessing the data
console.log(obj.name); // John
```

## Converting JavaScript Object to JSON

You can also convert JavaScript objects to JSON format using the JavaScript built-in JSON.stringify() function. For example,

```
// JavaScript object
const jsonData = { "name": "John", "age": 22 };

// converting to JSON
const obj = JSON.stringify(jsonData);

// accessing the data
console.log(obj); // “{"name":"John","age":22}"
```

## Use of JSON

JSON is the most commonly used format for transmitting data (data interchange) from a server to a client and vice-versa. JSON data are very easy to parse and use. It is fast to access and manipulate JSON data as they only contain texts.

JSON is language independent. You can create and use JSON in other programming languages too.