**Synchronous JavaScript**

Synchronous is a blocking architecture, so the execution of each operation depends on completing the one before it. Each task requires an answer before moving on to the next iteration.

* Sync is a single-thread, so only one operation or program will run at a time.
* Sync is blocking — it will only send the server one request at a time and wait for that request to be answered by the server.
* Sync is slower and more methodical.

function f1() {

  console.log("f1 execution");

}

function f2() {

  console.log("f2 execution");

}

function f3() {

  console.log("f3 execution");

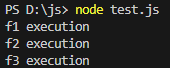
}

// Invoke the functions one by one

f1();

f2();

f3();



**Callback function in javascript?**

A callback function can be defined as a function passed into another function as a parameter. In other words, we can say that a function passed to another function as an argument is referred to as a callback function. The callback function runs after the completion of the outer function. It is useful to develop an asynchronous JavaScript code.

Example 1:

function greetings(username) {

  console.log('Welcome ' + username);

}

//the function declaration above is to be passed into the function below thereby making it a callback function.

function saveUserName(callback) {

  var username = 'Kamal Pratap';

  callback(username);

}

//the above function states the callback function as a parameter

saveUserName(greetings);

//the greetings function (the callback function), is initially declared, then invoked by passing it as an argument into another function.



Example 2:

// Main function

const mainFunction = (callback) => {

  setTimeout(() => {

      callback([2, 3, 4]);

  }, 2000)

  console.log('Function Executed');

}

// Add function

const add = (array) => {

  let sum = 0;

  for(let i of array) {

      sum += i;

  }

  console.log(sum);

}

// Calling main function

mainFunction(add);

****

**Asynchronous JavaScript**

Asynchronous is a non-blocking architecture, so the execution of one task isn’t dependent on another. Tasks can run simultaneously.

The asynchronous programming language focuses on the to enhance the application's performance. The callbacks can be used in such scenarios. We can analyze the asynchronous behavior of JavaScript by the below example:

* Async is multi-thread, which means operations or programs can run in parallel.
* Async is non-blocking, which means it will send multiple requests to a server.
* Async increases throughput because multiple operations can run at the same time.

function greet(){

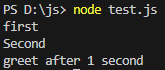
  console.log("greet after 1 second")

}

setTimeout(greet, 1000)

console.log("first")

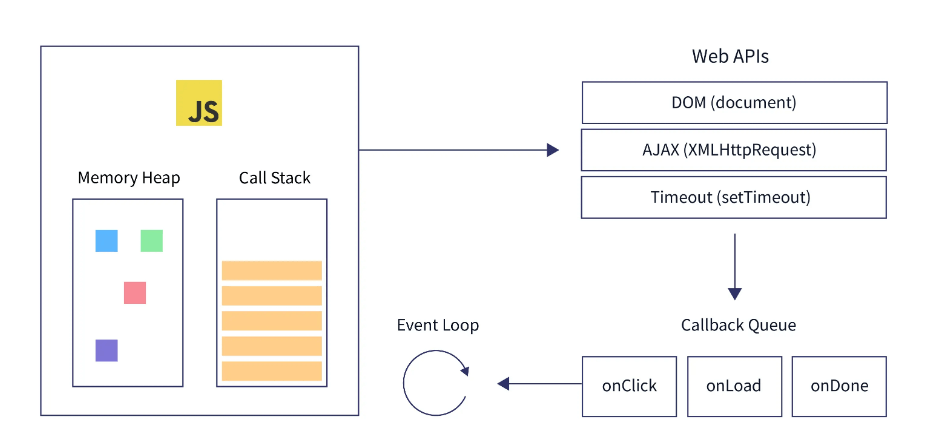
console.log("Second")



In JavaScript, setTimeout is an asynchronous function. Whenever we call the setTimeout function, it registers a callback function (greet in this case) to be executed after the specified delay. However, it does not block the execution of the subsequent code.

We need to understand the call stack and callback queue to handle the asynchronous events in JavaScript.

Consider the below image:



From the above image, a typical JavaScript engine consists of a heap memory and a call stack. The call stack executes all the code without waiting when pushed to the stack.

The heap memory is responsible for allocating the memory for objects and functions at runtime whenever they are needed.

Now, our browser engines consist of several web APIs such as DOM, setTimeout, console, fetch, etc., and the engine can access these APIs using the global window object. In the next step, some event loops play the role of gatekeeper that picks function requests inside the callback queue and pushes them into the stack. These functions, such as setTimeout, require a certain waiting time.

Now, let's go back to our example, the setTimeout function; when the function gets encountered, the timer gets registered in the callback queue. After this, the rest of the code is pushed into the call stack and gets executed once the function reaches its timer limit, it is expired, and the callback queue pushes the callback function, which has the specified logic and is registered in the timeout function. Thus, it will be executed after the specified time.

**Callback Hell Scenarios**

Now, we have discussed the callbacks, synchronous, asynchronous, and other relevant topic for the callback hell. Let's understand what callback hell is in JavaScript.

The situation when multiple callbacks are nested is known as the callback hell since its code shape looks like a pyramid, which is also called the "pyramid of the doom".

The callback hell makes it harder to understand and maintain the code. We can mostly see this situation while working in node JS. For example, consider the below example:

function print(i){

 console.log('This is call number '+i);

}

function fun1(callback){

  setTimeout(()=>{

      let i = 1 ;

      callback(i); i++ ;

      setTimeout(()=>{

          callback(i); i++;

          setTimeout(()=>{

            callback(i); i++ ;

            setTimeout(()=>{

                callback(i); i++ ;

                setTimeout(()=>{

                  callback(i); i++ ;

                  // .... and so on

                }, 1000)

            }, 1000)

          }, 1000)

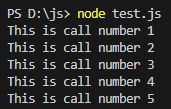
      }, 1000)

  }, 1000)

}

// Calling fun1 with print function as parameter

fun1(print);



**Escaping Callback Hell**

**1.Promises to the Rescue**

JavaScript Promise are easy to manage when dealing with multiple asynchronous operations where callbacks can create callback hell leading to unmanageable code. A [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) is an object representing the eventual completion or failure of an asynchronous operation.

In JavaScript, a Promise is an [object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_objects) that will produce a single value sometime in the future. If the promise is successful, it will produce a resolved value, but if something goes wrong then it will produce a reason why the promise failed. The possible outcomes here are similar to that of promises in real life.

let promise = new Promise(function(resolve, reject){

  //do something

});

JavaScript promises can be in one of three possible states.

* pending: This is the default state of a defined promise
* fulfilled:  This is the state of a successful promise
* rejected: This is the state of a failed promise

Example 1:

const promiseobj = new Promise((resolve, reject) => {

  const num = Math.random();

  console.log(num);

  if (num >= 0.5) {

    resolve("Promise is fulfilled!");

  }

  else {

    reject("Promise failed!");

  }

});

function handleResolve(result) {

  console.log(result);

}

function handleReject(result) {

  console.error(result);

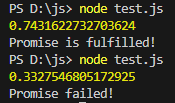
}

promiseobj.then(handleResolve, handleReject);

// Promise is fulfilled!

// or

// Promise failed!



Example 2:

let p = new Promise((resolve, reject) => {

  let isTrue = false;

  if (isTrue) {

    resolve('Success');

  } else {

    reject('Error');

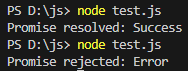
  }

});

p

.then(message => console.log(`Promise resolved: ${message}`))

.catch(message => console.log(`Promise rejected: ${message}`));



**2. Async/Await**

The [**async**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function) keyword gives you a simpler way to work with asynchronous promise-based code. Adding async at the start of a function makes it an async function.

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

// 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();



**Hoisting**

In [JavaScript](https://www.javatpoint.com/javascript-tutorial), Hoisting is a kind of default behavior in which all the declarations either variable declaration or function declaration are moved at the top of the scope just before executing the program's code.

**Function Hoisting**

Take a look at this code example:

function printHello() {

  console.log("hello")

}

printHello()

// hello

Here, we declare printHello, and we execute the function just after the line it was declared. No errors; everything works!

Now look at this example:

printHello()

// hello

function printHello() {

  console.log("hello")

}

Here, we execute printHello before the line the function was declared. And everything still works without errors. What happened here? **Hoisting**

**Event Bubbling and Capturing**

Event bubbling and capturing are two ways of event propagation in the HTML DOM, when an event occurs in an element inside another element, and both elements have registered a handle for that event. The event propagation mode determines in [which order the elements receive the event](http://www.quirksmode.org/js/events_order.html).

With bubbling, the event is first captured and handled by the innermost element and then propagated to outer elements.

With capturing, the event is first captured by the outermost element and propagated to the inner elements.

Capturing is also called "trickling", which helps remember the propagation order

<html>

<head>

<style>

  body {

    padding: 20px;

    background-color: pink;

  }

  div {

    padding: 20px;

    background-color: green;

    width: max-content;

  }

  span {

    display: block;

    padding: 20px;

    background-color: blue;

  }

</style>

<script>

  window.onload=function(){

  const body = document.getElementsByTagName("body")[0]

  const div = document.getElementsByTagName("div")[0]

  const span = document.getElementsByTagName("span")[0]

  const button = document.getElementsByTagName("button")[0]

  body.addEventListener('click', () => {

    console.log("body was clicked")

  })

  div.addEventListener('click', () => {

    console.log("div was clicked")

  })

  span.addEventListener('click', () => {

    console.log("span was clicked")

  })

  button.addEventListener('click', () => {

    console.log("button was clicked")

  })

}

</script>

</head>

<body>

  <div>

    <span>

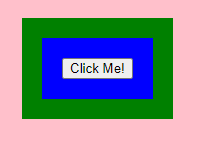
      <button>Click Me!</button>

    </span>

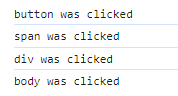
  </div>

</body>

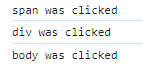
</html>



When i click on button



When I click on span



When I click on div



**How to Stop Event Bubbling**

Event Bubbling is a default behaviour for events. But in some cases, you might want to prevent this.

To prevent event bubbling, you use the **stopPropagation** method of the event object.

body.addEventListener('click', () => {

  console.log("body was clicked")

})

div.addEventListener('click', () => {

  console.log("div was clicked")

})

span.addEventListener('click', (event) => {

  event.stopPropagation()

  console.log("span was clicked")

})

button.addEventListener('click', () => {

  console.log("button was clicked")

})

