**Introduction to Asynch Programming**

***Synchronous* programming**:-In this model, things happen one at a time. When you call a function that performs a long-running action, it returns only when the action has finished and it can return the result. Means person would have to wait for the person before them to finish their task before starting their own. This stops your program for the time the action takes.

**Here's an example of synchronous code in JavaScript:**

// Define three functions

**function firstTask() { OUTPUT**

**console.log("Task 1");**

**}**

**function secondTask() {**

**console.log("Task 2");**

**}**

**function thirdTask() {**

**console.log("Task 3");**

**}**

// Execute the functions

**firstTask();**

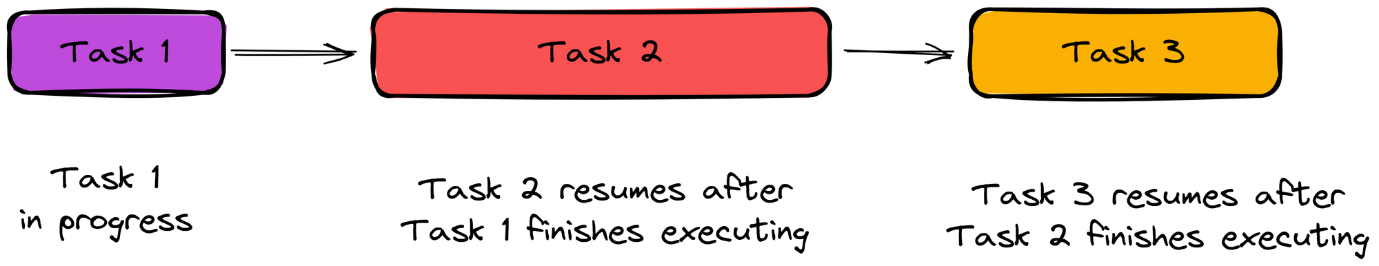
**secondTask();**

**thirdTask();**

**"Task 1"**

**"Task 2"**

**"Task 3"**



***Problem with Synchronous Programming:-***

In a synchronous environment, where the request function returns only after it has done its work, the easiest way to perform this task is to make the requests one after the other. This has the drawback that the second request will be started only when the first has finished. The total time taken will be at least the sum of the two response times.

**for example:-**

let's say that a synchronous program performs a task that requires waiting for a response from a remote server. The program will be stuck waiting for the response and cannot do anything else until the response is returned. This is known as blocking, and it can lead to an application appearing unresponsive or "frozen" to the user.

**function someLongRunningFunction() {**

**let start = Date.now();**

**while (Date.now() - start < 5000) {**

// do nothing

**}**

**return "Hello";**

**}**

**console.log('Starting...');**

**let result = someLongRunningFunction();**

**console.log(result);**

**console.log('...Finishing');**

***Explanation for above code:-***

* The program starts by logging *"Starting..."* to the console.
* Then it calls the **someLongRunningFunction()**, which simulates a long-running task that takes 5 seconds to complete. This function will block the execution of the rest of the program while it runs.
* Once the function completes, it will return *"Hello"*, and the program will log it on the console.
* Finally, the program will log *"Finishing"* to the console.

***Problem Analysis:-***

During the 5 seconds that **someLongRunningFunction()** is being executed, the program will be blocked, become unresponsive, and be unable to execute the next line of code. This can cause the program to take a long time to complete and make the application unresponsive to the user.

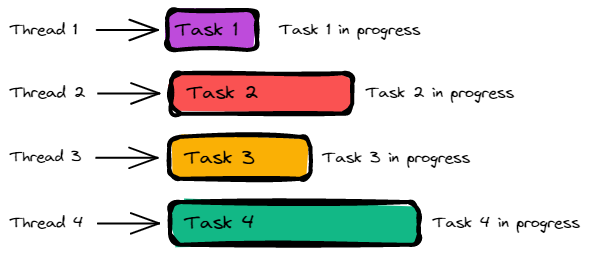
***Solution:-***

*if the program is executed asynchronously, it will continue to run the next line of code instructions rather than becoming blocked. This will enable the program to remain responsive and execute other code instructions while waiting for the timeout to complete.*

***What is Asynchronous*** **programming**:-

* An *asynchronous* model allows multiple things to happen at the same time.
* When you start an action, your program continues to run. When the action finishes, the program is informed and gets access to the result (for example, the data read from disk).
* Means everyone can start and work on their tasks simultaneously without waiting for the others to finish.
* This approach can greatly improve the performance and responsiveness of a program.
* for example, while a program retrieves data from a remote server, it can continue to execute other tasks such as responding to user inputs.

Asynchronoicity makes expressing programs that do not fit the straight-line model of control easier, but it can also make expressing programs that do follow a straight line more awkward.



**Example:-**

Here's an example of an asynchronous program using the **setTimeout** method:

**console.log("Start of script"); OUTPUT**

**setTimeout(function() {**

**console.log("First timeout completed");**

**}, 2000);**

**console.log("End of script");**

**Start of script**

**End of script**

**First timeout completed**

**Explanation:-**

* The **setTimeout** method executes a function after a specified time.
* The function passed to **setTimeout** will be executed asynchronously, which means that the program will continue to execute the next line of code without waiting for the timeout to complete.
* As we can see, **console.log("First timeout completed")**will be executed after 2 seconds. Meanwhile, the script continues to execute the next code statement and doesn't cause any "blocking" or "freezing" behaviour.

**NOTE:-** In JavaScript, asynchronous programming can be achieved through a variety of techniques. One of the most common methods is the use of **callbacks.**

## Event handlers:- are really a form of asynchronous programming. you provide a function (the event handler) that will be called, not right away, but whenever the event happens. If "the event" is "the asynchronous operation has completed", then that event could be used to notify the caller about the result of an asynchronous function call.

## Callbacks

## Callbacks:- An event handler is a particular type of callback. A callback is just a function that's passed into another function, with the expectation that the callback will be called at the appropriate time. Callbacks used to be the main way asynchronous functions were implemented in JavaScript.However, callback-based code can get hard to understand when the callback itself has to call functions that accept a callback.

## Example 1:- Normal function

## <script>

## const funA = () => {

## document.write("hey<br>");

## }

## const funB = () => {

## document.write("hey function b");}

## funA();funB();</script>

## How to Use a Callback Function:-

* Let's say you want to plan a birthday party for your child. You have to invite the guests, order a cake, and plan the games. But you also want to hire a clown to entertain the guests. You can only have the clown come to the party once all the other party arrangements are done, and the guests have arrived.
* So, you tell the clown to come to the party only after you have notified him that the guests have arrived. In this case, the clown represents a callback function, and the "guests arriving" represents the function that has to complete execution before the callback can be executed.

**In code, a callback function is a function that is passed as an argument to another function, and it is executed after the first function has finished running.** It's commonly used in JavaScript to handle asynchronous operations like fetching data from a server, waiting for a user's input, or handling events.

**Example 2:-**

<script>

const funA = (name, arg2) => {

document.write(`hey ${name}. ! How are you`);

arg2(); // this is now a callback function

}

const funB = () => {

document.write("hey function b");

}

funA("preenu",funB);

</script>

**Example 3:**- **How you can use a callback function to handle an asynchronous operation**

// Declare function

**function fetchData(callback) {**

**setTimeout(() => {**

**const data = {name: "John", age: 30};**

**callback(data);**

**}, 3000);**

**}**

// Execute function with a callback

**fetchData(function(data) {**

**console.log(data);**

**});**

**console.log("Data is being fetched...");**

**Explanation:-**

* We have a function called **fetchData** that uses the **setTimeout** method to simulate an asynchronous operation. The function takes a **callback** as an argument.
* The **callback** function is then passed the data retrieved by the function after the timeout has been completed.

The**setTimeout** method is used to execute the **callback** after a specified time (in this case, 3 seconds). The **callback** will be executed asynchronously, which means that the program will continue to execute the next line of code without waiting for the timeout to complete.

This is the core concept of asynchronous programming. The script doesn't wait for the asynchronous operation to complete. It just continues to execute the next instruction.

## What is Callback Hell?

Callbacks provide a useful way to handle asynchronous operations. However, when many callbacks are nested, the code can be complex and hard to read and understand.

This happens when you chain multiple callbacks together, one after the other, creating a pyramid-like structure of indentation called callback hell, also known as the "Pyramid of Doom".

**Example:-**

getData(function(a) {

getMoreData(a, function(b) {

getEvenMoreData(b, function(c) {

getEvenEvenMoreData(c, function(d) {

getFinalData(d, function(finalData) {

console.log(finalData);

});

});

});

});

});

**Exaplanation:-**

* The **getData** function takes a **callback** as an argument and is executed after data is retrieved.
* The callback function then takes the data and calls the **getMoreData** function, which also takes a **callback** as an argument, and so on.

**NOTE:-**This nesting of callbacks can make the code difficult to maintain, and the indentation makes it even harder to see the overall structure of the code.  most modern asynchronous APIs don't use callbacks.

To avoid callback hell, you can use a more modern way of handling async operations known as promises. Promises provide a more elegant way of handling the asynchronous flow of a program compared to callback functions.

**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); |

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.

|  |
| --- |
| **Example 2:-**  <html><head><script>  function func1(){  return new Promise(function(resolve,reject) {  setTimeout(() => {  const error = false;  if (!error) {  document.write("Function: has succesfully worked")  resolve();  }  else{  document.write("Function: has rejected")  reject('sorry Not Worked');  }  },2000);  })  }  func1().then(function(){  document.write("Promise! Thanks for HElp")  }).catch(function(error){  document.write("Preenu! Try Again " + error)  })  </script><body></body></html> |

Example 3:-

|  |
| --- |
| <script>  function func1()  {  return new Promise(function(resolve,reject) {  setTimeout(() => {  const error = true;  if (!error) {  document.write("Function: has succesfully worked")  resolve();  }  else  {  document.write("Function: has rejected")  reject('sorry Not Worked');  }  },2000);  })  }  func1().then(function(){  document.write("Promise! Thanks for HElp")  }).catch(function(error){  document.write("Preenu! Try Again " + error)  })  </script> |

**Example 4:- promise with callback**

<script>

const students = [

{name: "preenu" ,subject: "javascript"},

{name: "rohan" ,subject: "Python"} ]

function enrollStudent(student){

return new Promise(function(resolve,reject){

setTimeout(function() {

students.push(student);

document.write("student has been enrolled");

const error=false;

if(!error){

resolve();

}

else

{

reject();

}

},1000);

})

}

function getStudent(){

setTimeout(function() {

let str = "";

students.forEach(function(student){

str += `<li> ${student.name}</li>`

});

document.getElementById('students').innerHTML = str;

document.write("students have been fetched");

},5000);

}

let newStudent = { name: "preeti" ,subject: "mL"}

enrollStudent(newStudent).then(getStudent).catch(function(){

document.write("some error")

});

</script>

**JavaScript Nested Function**

In JavaScript, a function can also contain another function. This is called a nested function.

**Example 1:- // nested function example**

|  |
| --- |
| <html>  <head>  <title>Nested functions</title>  </head>  <body><script>  // outer function  function greet(name) {  // inner function  function displayName() {  console.log('Hi' + ' ' + name);  }  // calling inner function  displayName();  }  // calling outer function  greet('Javascript');  </script></body></html> |

**Returning a Function**

you can also return a function within a function.

|  |
| --- |
| <html>  <head>  <title>Nested functions</title>  </head>  <body>  <script>  function greet(name) {  function displayName() {  console.log('Hi' + ' ' + name);  }  // returning a function  return displayName;  }  const g1 = greet('Javascript');  console.log(g1); // returns the function definition  g1(); // calling the function  </script>  </body></html> |

**Explanation:-**

In the above program, the greet() function is returning the displayName function definition.

Here, the returned function definition is assigned to the g1 variable. When you print g1 using console.log(g1), you will get the function definition.

To call the function stored in the g1 variable, we use g1() with parentheses.

**JavaScript Closures**

* In JavaScript, closure provides access to the outer scope of a function from inside the inner function, even after the outer function has closed.
* Closure is action that is inner function can have access to the outer function variables as well as all the global variables
* A closure is a function having access to the parent scope. It preserve the data from outside.
* For every closure we have three scopes:-
* Local scope (OWN scope)
* Outer functions scope
* Global scope

**Example 1:-**

|  |
| --- |
| <html>  <head>  <title>Nested functions</title>  </head>  <body>  <script>  var i =10;  **function result()**  {  var j=20;  document.write(j+"<br>");  // document.write(i+"<br>");  }  **result();**  </script>  </body></html> |

**Explanation:-**

Here we are accessing the variable “j” that is part of result function so that is accessible but if we want to access the variable “I” also that would also be accessible inside this function as it is global variable so this was a simple example that illustrate the closure here also.

**Example 2:-**

|  |
| --- |
| <html>  <head>  <title>Nested functions</title>  </head>  <body><script>  function outerfun()  {  var j="hello im part of outer function";  document.write(j+"<br>");  function innerfun()  {  var k="im part of inner function";  document.write(k+"<br>");  //document.write(j+"<br>");  }  innerfun()  document.write(k+"<br>");  }  outerfun();  </script></body></html> |

**Explanation:-**

Here we are having two function one is outer and one is inner so inner function can access the variable of outer also that is closure but outer can not access the methods and variables of inner if we will try to access them will caught with the error.

* The return statement does not execute the inner function- function is executed only when followed by () ,but rather the return statement returns the entire body of the function

**Example 3:-**

|  |
| --- |
| <html><head><title>Nested functions</title></head>  <body><script>  const outerfun = (a) => {  let j=10;  const innerfun = () =>  {  var k=a+j;  document.write(`the sum of two no is ${k}.`);  }  innerfun()  }  outerfun(5);  </script></body></html> |

**Example 4:-**

|  |
| --- |
| <html>  <head>  <title>Nested functions</title>  </head>  <body>  <script>  // closure example  function calculate(x) {  function multiply(y) {  return x \* y;  }  return multiply;  }  const multiply3 = calculate(3);  const multiply4 = calculate(4);  console.log(multiply3); // returns calculate function definition  console.log(multiply3()); // NaN  console.log(multiply3(6)); // 18  console.log(multiply4(2)); // 8  </script></body></html> |

**Explanation:-**

In the above program, the calculate() function takes a single argument x and returns the function definition of the multiply() function. The multiply() function takes a single argument y and returns x \* y.

Both multiply3 and multiply4 are closures.

The calculate() function is called passing a parameter x. When multiply3(6) and multiply4(2) are called, the multipy() function has access to the passed x argument of the outer calculate() function.

**IIFE (immediately invoked Function Expression)**

IIFE is a javascript function that runs as soon as it is defined.

It is a design pattern which is also known as Self-Executing Anonymous Function and contains two major parts. The First is the anonymous function with lexical scope enclosed within the Grouping Operator ().

This prevents accessing variable within the IIFE idiom as well as polluting the global scope.

The second part is creating the immediately executing function expression(), through which the js engine will directly interpret the function.

Ex :-

(function () {

document.write(“Today is sunny day”);

} )

();

(function (a,b) { document.write(a + “ “ +b); } )(4,4);

**Important points:-**

* Avoid creating global variables and functions.
* Scope is limited to that particular function only