**Q1: what is callback function and callback hell**

**Ans :**A callback is a function that is passed as an argument to another function that executes the callback based on the result. They are basically functions that are executed only after a result is produced. Callbacks are an important part of asynchronous JavaScript.

// Main function

const mainFunction = (callback) => {

setTimeout(() => {

callback([2, 3, 4]);

}, 2000)

}

// Add function

const add = (array) => {

let sum = 0;

for(let i of array) {

sum += i;

}

console.log(sum);

}

// Calling main function

mainFunction(add);

Callback Hell is essentially nested callbacks stacked below one another forming a pyramid structure. Every callback depends/waits for the previous callback, thereby making a pyramid structure that affects the readability and maintainability of the code.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta http-equiv="X-UA-Compatible"

content="IE=edge">

<meta name="viewport" content=

"width=device-width, initial-scale=1.0">

<title>Callback Hell</title>

<style>

\* {

padding: none;

margin: none;

box-sizing: border-box;

}

.word {

color: #308d46;

font-size: 4rem;

transition: all .5s ease-in;

margin: 0 5px;

transform: translateY(3.8rem);

opacity: 0;

}

body {

display: flex;

justify-content: center;

align-items: center;

width: 95vw;

height: 95vh;

}

.container {

overflow: hidden;

display: flex;

flex-direction: row;

}

.animate {

opacity: 1;

transform: translateY(0);

}

</style>

</head>

<body>

<div class="container">

<h2 class="word">Geeks</h2>

<h2 class="word">For</h2>

<h2 class="word">Geeks</h2>

</div>

</body>

<script>

let words = document.querySelectorAll(".word");

const animateAll = (animate) => {

setTimeout(() => {

animate(words[0]);

setTimeout(() => {

animate(words[1]);

setTimeout(() => {

animate(words[2]);

}, 1000)

}, 1000)

}, 1000)

}

const animate = (word) => {

word.classList.add("animate");

}

animateAll(animate);

</script>

</html>

Q2: Explain how to iterate an object in javascript?

There are two methods to iterate over an object which are discussed below:

**Method 1**

**Using for in loop:** The properties of the object can be iterated over using a for..in loop. This loop is used to iterate over all non-Symbol iterable properties of an object. Some objects may contain properties that may be inherited from their prototypes. The **hasOwnProperty()** method can be used to check if the property belongs to the object itself. The value of each key of the object can be found by using the key as the index of the object.

Syntax: for (let key in exampleObj) {

if (exampleObj.hasOwnProperty(key)) {

value = exampleObj[key];

console.log(key, value);

}

}

Example:

<!DOCTYPE html>

<html>

<head>

<title>

How to iterate over

a JavaScript object?

</title>

</head>

<body>

<h1 style="color: green">

GeeksforGeeks

</h1>

<b>

How to iterate over

a JavaScript object?

</b>

<p>

Click on the button to iterate

through the javascript object.

</p>

<p>

Check the console

for the output

</p>

<button onclick="iterateObject()">

Iterate Object

</button>

<script type="text/javascript">

function iterateObject() {

let exampleObj = {

book: "Sherlock Holmes",

author: "Arthur Conan Doyle",

genre: "Mystery"

};

for (let key in exampleObj) {

if (exampleObj.hasOwnProperty(key))

{

value = exampleObj[key];

console.log(key, value);

}

}

}

</script>

</body>

</html>

**Q3 : Explain about Promises and create a custom promise with simple Example.**

A JavaScript Promise object contains both the producing code and calls to the consuming code

Promise Syntax:

let myPromise = new Promise(function(myResolve, myReject) {  
// "Producing Code" (May take some time)  
  
  myResolve(); // when successful  
  myReject();  // when error  
});  
  
// "Consuming Code" (Must wait for a fulfilled Promise)  
myPromise.then(  
  function(value) { /\* code if successful \*/ },  
  function(error) { /\* code if some error \*/ }  
);

we can use promise like this:

myPromise.then(  
  function(value) { /\* code if successful \*/ },  
  function(error) { /\* code if some error \*/ }  
);

Example:function myDisplayer(some) {  
  document.getElementById("demo").innerHTML = some;  
}  
  
let myPromise = new Promise(function(myResolve, myReject) {  
  let x = 0;  
  
// The producing code (this may take some time)  
  
  if (x == 0) {  
    myResolve("OK");  
  } else {  
    myReject("Error");  
  }  
});  
  
myPromise.then(  
  function(value) {myDisplayer(value);},  
  function(error) {myDisplayer(error);}  
);

Q4:**What is implicit type coercion in Javascript?**

**Type Coercion** refers to the process of automatic or implicit conversion of values from one data type to another. This includes conversion from Number to String, String to Number, Boolean to Number etc. when different types of operators are applied to the values.

In case the behavior of the implicit conversion is not sure, the constructors of a data type can be used to convert any value to that datatype, like the **Number()**, **String()**or **Boolean()** constructor.

**1. String to Number Conversion:** When any string or non-string value is added to a string, it always converts the non-string value to a string implicitly. When the string ‘Rahul’ is added to the number 10 then JavaScript does not give an error. It converts the number 10 to string ’10’ using coercion and then concatenates both the strings. Some more examples are shown below.

**Example:**

<!DOCTYPE html>

<html>

<head>

<title>

What is Type Coercion

in JavaScript?

</title>

</head>

<body>

<h1 style="color: green;">

GeeksforGeeks

</h1>

<h2>Type Coercion</h2>

<h3>Output</h3>

<script>

// The Number 10 is converted to

// string '10' and then '+'

// concatenates both strings

var x = 10 + '20';

var y = '20' + 10;

// The Boolean value true is converted

// to string 'true' and then '+'

// concatenates both the strings

var z = true + '10';

document.write(x);

document.write("<br>");

document.write(y);

document.write("<br>");

document.write(z);

</script>

</body>

</html>

**2. String to Number Conversion:** When an operation like subtraction (-), multiplication (\*), division (/) or modulus (%) is performed, all the values that are not number are converted into the number data type, as these operations can be performed between numbers only. Some examples of this are shown below.

Example: <!DOCTYPE html>

<html>

<head>

<title>

What is Type Coercion

in JavaScript?

</title>

</head>

<body>

<h1 style="color: green;">

GeeksforGeeks

</h1>

<h2>Type Coercion</h2>

<h3>Output</h3>

<script>

// The string '5' is converted

// to number 5 in all cases

// implicitly

var w = 10 - '5';

var x = 10 \* '5';

var y = 10 / '5';

var z = 10 % '5';

document.write(w);

document.write("<br>")

document.write(x);

document.write("<br>")

document.write(y);

document.write("<br>")

document.write(z);

</script>

</body>

</html>

**3. Boolean to Number:** When a Boolean is added to a Number, the Boolean value is converted to a number as it is safer and easier to convert Boolean values to Number values. A Boolean value can be represented as 0 for ‘false’ or 1 for ‘true’. Some examples of this are shown below.

Example: <!DOCTYPE html>

<html>

<head>

<title>

What is Type Coercion

in JavaScript?

</title>

</head>

<body>

<h1 style="color: green;">

GeeksforGeeks

</h1>

<h2>Type Coercion</h2>

<h3>Output</h3>

<script>

// The Boolean value true is

// converted to number 1 and

// then operation is performed

var x = true + 2;

// The Boolean value false is

// converted to number 0 and

// then operation is performed

var y = false + 2;

document.write(x);

document.write("<br>")

document.write(y);

</script>

</body>

</html>

**4. The Equality Operator:** The equality operator (==) can be used to compare values irrespective of their type. This is done by coercing a non-number data type to a number. Some examples of this are shown below:

Example: <!DOCTYPE html>

<html>

<head>

<title>

What is Type Coercion

in JavaScript?

</title>

</head>

<body>

<h1 style="color: green;">

GeeksforGeeks

</h1>

<h2>Type Coercion</h2>

<h3>Output</h3>

<script>

// Should output 'true' as string '10'

// is coerced to number 10

var x = (10 == '10');

// Should output 'true', as boolean true

// is coerced to number 1

var y = (true == 1);

// Should output 'false' as string 'true'

// is coerced to NaN which is not equal to

// 1 of Boolean true

var z = (true == 'true');

document.write(x);

document.write("<br>");

document.write(y);

document.write("<br>");

document.write(z);

</script>

</body>

</html>

Q5:**Explain pass by value and pass by reference in javascript?**

**Pass By Value:** In Pass by value, function is called by directly passing the value of the variable as an argument. So any changes made inside the function does not affect the original value.

In Pass by value, parameters passed as an arguments create its **own copy** So any changes made inside the function is made to the copied value not to the original value.

Example:

function Passbyvalue(a, b) {

let tmp;

tmp = b;

b = a;

a = tmp;

console.log(`Inside Pass by value

function -> a = ${a} b = ${b}`);

}

let a = 1;

let b = 2;

console.log(`Before calling Pass by value

Function -> a = ${a} b = ${b}`);

Passbyvalue(a, b);

console.log(`After calling Pass by value

Function -> a =${a} b = ${b}`);

**Pass by Reference:** In Pass by Reference, Function is called by directly passing the reference/address of the variable as an argument. So changing the value inside the function also change the original value. In JavaScript **array and Object**follows pass by reference property.

In Pass by reference, parameters passed as an arguments does not create its own copy, it refers to the original value so changes made inside function affect the original value.

let us take an example to understand better.

Example: function PassbyReference(obj) {

let tmp = obj.a;

obj.a = obj.b;

obj.b = tmp;

console.log(`Inside Pass By Reference

Function -> a = ${obj.a} b = ${obj.b}`);

}

let obj = {

a: 10,

b: 20

}

console.log(`Before calling Pass By Reference

Function -> a = ${obj.a} b = ${obj.b}`);

PassbyReference(obj)

console.log(`After calling Pass By Reference

Function -> a = ${obj.a} b = ${obj.b}`);

Q6: **what is the difference between flex and Grid explain with example?**

[Grid](https://www.geeksforgeeks.org/css-grid-property/)**:** CSS Grid Layout, is a two-dimensional grid-based layout system with rows and columns, making it easier to design web pages without having to use floats and positioning. Like tables, grid layout allow us to align elements into columns and rows.

To get started you have to define a container element as a grid with **display: grid,** set the column and row sizes with grid-template-columns and grid-template-rows, and then place its child elements into the grid with grid-column and grid-row.

Example:

<!DOCTYPE html>

<html lang="en">

<head>

<style>

.main{

display: grid;

display: grid;

grid: auto auto / auto auto auto auto;

grid-gap: 10px;

background-color: green;

padding: 10px;

}

.gfg {

background-color: rgb(255, 255, 255);

text-align: center;

padding: 25px 0;

font-size: 30px;

}

</style>

</head>

<body>

<h2 style="text-align: center;">Welcome To GeeksForGeeks </h2>

<div class="main">

<div class="gfg">Home</div>

<div class="gfg">Read</div>

<div class="gfg">Write</div>

<div class="gfg">About Us</div>

<div class="gfg">Contact Us</div>

<div class="gfg">Privacy Policy</div>

</div>

</body>

</html>

[Flexbox](https://www.geeksforgeeks.org/introduction-to-css-flexbox/)**:** The CSS Flexbox offers a one-dimensional layout. It is helpful in allocating and aligning the space among items in a container (made of grids). It works with all kinds of display devices and screen sizes.

To get started you have to define a container element as a grid with **display: flex;**

**<!DOCTYPE html>**

**<html lang="en">**

**<head>**

**<style>**

**.main{**

**display: flex;**

**display: flex;**

**grid: auto auto / auto auto auto auto;**

**grid-gap: 10px;**

**background-color: green;**

**padding: 10px;**

**}**

**.gfg {**

**background-color: rgb(255, 255, 255);**

**text-align: center;**

**padding: 25px 0;**

**font-size: 30px;**

**}**

**</style>**

**</head>**

**<body>**

**<h2 style="text-align: center;">Welcome To GeeksForGeeks </h2>**

**<div class="main">**

**<div class="gfg">Home</div>**

**<div class="gfg">Read</div>**

**<div class="gfg">Write</div>**

**<div class="gfg">About Us</div>**

**<div class="gfg">Contact Us</div>**

**<div class="gfg">Privacy Policy</div>**

**</div>**

**</body>**

**</html>**

**Q7:Explain about Currying with Example?**

[Currying](https://en.wikipedia.org/wiki/Currying) is an advanced technique of working with functions. It’s used not only in JavaScript, but in other languages as well.

Currying is a transformation of functions that translates a function from callable as f(a, b, c) into callable as f(a)(b)(c).

Currying doesn’t call a function. It just transforms it.

Let’s see an example first, to better understand what we’re talking about, and then practical applications.

We’ll create a helper function curry(f) that performs currying for a two-argument f. In other words, curry(f) for two-argument f(a, b) translates it into a function that runs as f(a)(b):

Example:

function curry(f) { // curry(f) does the currying transform

return function(a) {

return function(b) {

return f(a, b);

};

};

}

// usage

function sum(a, b) {

return a + b;

}

let curriedSum = curry(sum);

alert( curriedSum(1)(2) ); // 3

As you can see, the implementation is straightforward: it’s just two wrappers.

* The result of curry(func) is a wrapper function(a).
* When it is called like curriedSum(1), the argument is saved in the Lexical Environment, and a new wrapper is returned function(b).
* Then this wrapper is called with 2 as an argument, and it passes the call to the original sum.

More advanced implementations of currying, such as [\_.curry](https://lodash.com/docs#curry) from lodash library, return a wrapper that allows a function to be called both normally and partially:

Example:

function sum(a, b) {

return a + b;

}

let curriedSum = \_.curry(sum); // using \_.curry from lodash library

alert( curriedSum(1, 2) ); // 3, still callable normally

alert( curriedSum(1)(2) ); // 3, called partially

**Q8:Explain about Object Prototype and add revese String method to string class to reverse a given String?**

All JavaScript objects inherit properties and methods from a prototype.

date objects inherit from Date.prototype

Array objects inherit from Array.prototype

Person objects inherit from Person.prototype

The Object.prototype is on the top of the prototype inheritance chain:

Date objects, Array objects, and Person objects inherit from Object.prototype.

**Example:**

**Object.defineProperty(String.prototype, 'reverseString', {**

**function reverseString(str2) {**

**var newString = "";**

**for (var i = str2.length - 1; i >= 0; i--) {**

**newString += str2[i];**

**}**

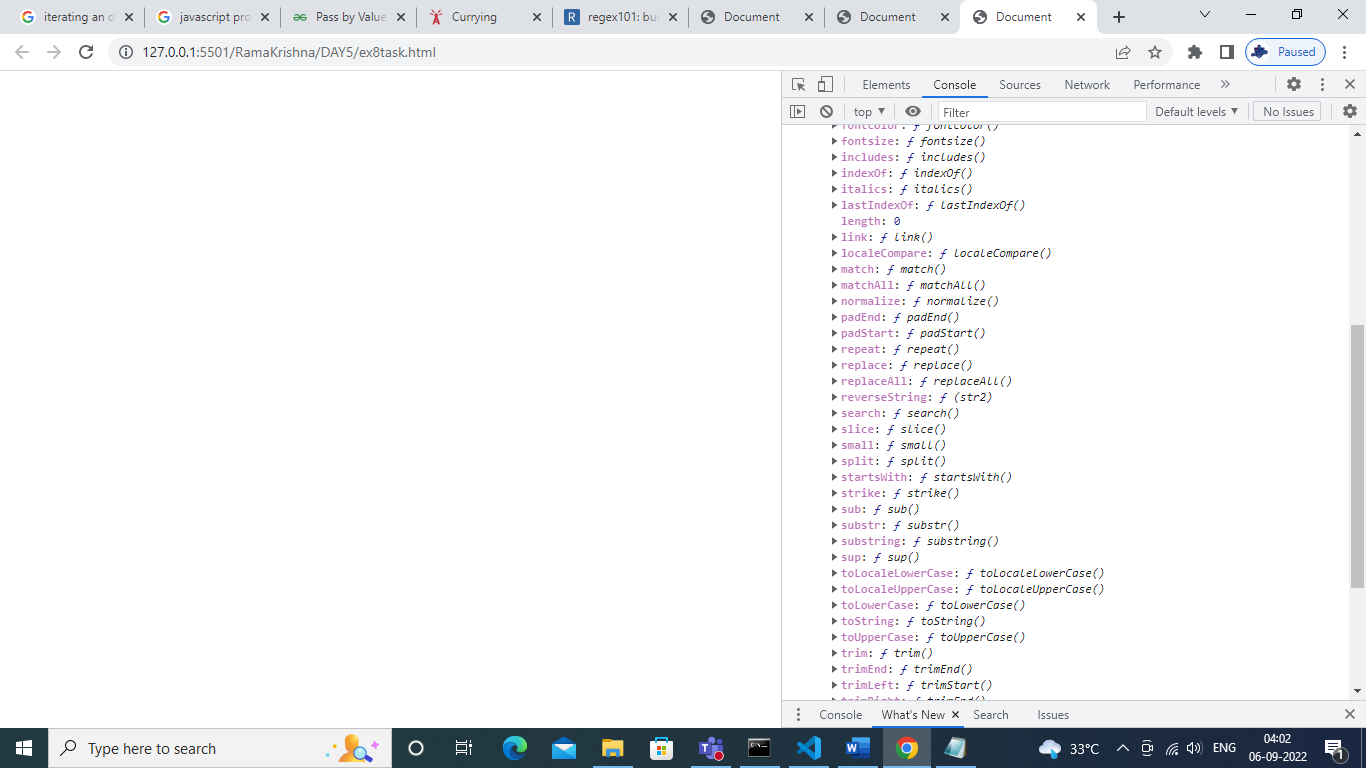
**return newString;**

**});**

**let str2 = new String("RAMAKRISHNA");**

**console.log(reverseString(str2));**

**ReverseString method in String Class**



Q9: **what are the differences between javascript and typescript?**

Javascript Typescript

|  |  |
| --- | --- |
| It is directly run on the browser. | It is not directly run on the browser. |
| JavaScript source file is in ".js" extension. | TypeScript source file is in ".ts" extension |
| In this, number, string are the objects. | In this, number, string are the interf |
| JavaScript doesn't support generics. | TypeScript supports generics. |
| It doesn't support optional parameters. | It supports optional parameters |
| It is just a scripting language. | It supports object-oriented programming concept like classes, interfaces, inheritance, generics . |
| JavaScript doesn't support modules. | TypeScript gives support for modules |
| function addNumbers(a, b) {  return a + b;  }  var sum = addNumbers(15, 25);  document.write('Sum of the numbers is: ' + sum); | function addNumbers(a, b) {  return a + b;  }  var sum = addNumbers(15, 25);  console.log('Sum of the numbe  is ‘ + sum); |

Q10:How to access data from one script block to another script block in javascript?