JavaScript...

* Programming.

- It is way to talk to computers.
- It is a process of creating a set of instructions that tell a computer how to perform a task.

* ECMA Script

- It is a standard on which Javascript is based.
- It is a standard for scripting languages including JavaScript, JScript & ActionScript.

* JavaScript. (Prototype based - lang.)

- It is a scripting language for web pages.
- It is a language which is used to give instructions to the computer.

* Diff. bet^n ECMA Script & JS.

- ECMA Script defines what a scripting language should look like. while JavaScript is an actual implementation of that specifications.

* Our 1st JS Code (Lect 1)

Console.log is used to log (Print) a message to the console.

```
Console.log ("Ankit Pawar");
```

* Variables in JS.

- Variables are the containers for storing the data values.
- It is casesensitive (a & A are diff^n)

* Let, Const & Var.

Var : Variables can be redeclared & updated
A global scope variable.

let : Variable cannot be redeclared but can be updated. Block Scope Variable.

Const : Variable cannot be redaclared & updatr. Block Scope Variable.

* Data types in JS.

- Data types is type of data which variable can be hold.

* Primitives - Number, String, Boolean, Undefined, Null, BigInt, Symbol.

- Non-Primitive - Objects, Arrays, etc

* Comments in JS.

- Part of code which is not executed.

```
// Singleline
/* Multiline */
```

* Operators in JS.

Used to perform some operations on data.

* Arithmetic Operator.

+, -, *, /, modulus (%), Exponentiation (**)
Increment (++), decrement (--).

* Assignment Operator.

=, +=, -=, *=, %=, **=.

```
a -=4  // a = a-4
```

* Comparison Operator.

Equal to (==), Equal to & type (===),
Not equal to (!=), Not equal to & type (!==).
>, >=, <, <=,

* Logical Operator.

AND (&&), OR (||), Not (!).

* Conditional Statements. (1 of 2)

- To implement some condition in the code

* if statement.

```
let color;
if (mode === "dark-mode")
{
    color = "black";
}
```

* if-else statement

```
let color;
if (mode === "dark-mode")
  { color = "black";
  }
else {
    color = "white";
}
```

* else-if statement

```
let age = 17;
if (age < 18) {
   Console.log ("junior");
}
else if (age > 60) {
}
else {
  }
```

* Ternary Operators.

condition ? true output : false output

age > 18 ? "adult" : "not adult";

```
let age = 25;
let result = age >= 18 ? "adult" : "not adult";
console.log (result);
```

* Switch statements

* Loops in JS

loops are used to execute a piece of code again & again.

* for loop

```
for (let i = 1; i <= 5; i++)
{
   console.log (i);
}
```

* While loop.

* do-while loop

* for-of loop

```
for (let val of str or)
{
}
```

```
let str = "Ankit";
for (let i of str) {
  console.log("i=", i);
}
```

x for-in loop.

```
for (let key in obj/var)
{
}
```

```
let student = {
name : "Ankit";
age : 20,
cpga : 7.5,
isPass : true,
};
for (let key in student)
{ console.log (key);
}
```

* Strings in JS. (immutable) (Unit 3)

- String is a sequence of characters used to represent text

- Create String

```
let str = "Ankit";
```

- String length

str.length

- String Indices

str[0], str[1], str[2]

* Template literals in JS

- It is a way to embedded expressions in strings (``)

`this is a template literal`.

- String Interpolation

- To create strings by doing substitution of placeholders.

`String text ${expression} string text`.

* Escape character

\n → next line
\t → tab line

- Strings methods in JS.

- There are built-in fn to manipulate a string.

- str.toUpperCase()
- str.toLowerCase()

- str.trim() // removes whitespace
- str.slice (start, end?)
  // return part of string
- str1.concat (str2)
- str.replace (search val, new val)
- str.charAt (idx)

* Arrays in JS. (mutable) (tut 4)

- It is a collection of items of similar type stored in contiguous memory location.

- Create Array

let marks = [99, 99, 99, 100];
0 1 2 3 ← index.

- looping over an array.
Print all elements of an array

- Arrays Methods

Push () : // add

Pop () : // delete

toString() : // Convert array to string

Concat () : // joining

unshift () : // add to start.

- shift (): // delete from start.

eg.

~~let heroes =~~
let marvelHeros = ["thor", "spiderman"];
let dcHeroes = ["superman"];
let heroes = marvelHeroes.concat (dcHeroes);

- slice (): // returns a piece of array
  slice (startidx, endidx)

- splice (): // change original array
  (add, remove, replace)

splice (startIdx, delcount, newEl1 --)

- eg [1, 2, 3, 4, 5, 6, 7]

splice [2, 2, 101, 102]

[1, 2, 101, 102, 5, 6]

* functions in JS. (tut 5)

- It is a block of code that performs a specific task which is runs after is called.

- fn Defn

function functName () {

}

```
function funcName (Para1 , Para2) {

}
```

- Function Call

```
functionName ();
```

* Arrow functions

Compact way of writing a fn.

```
const funcName = (Para1 , Para2, ...) =>
    {

    }
```

* Array Methods.

* Map. (tut 6)

- Creates a new array with the results of some operation. The value its callback returns are used to form new array.

```
arr.map (callbackfnx (value , index , array))

let newArr = arr.map ((val) => {
    return val*2;
})
```

* filter.

creates a new array of elements that give true for a condition / filter.

e.g. all even elements.

```
let newArr = arr.filter (val) => {
    return val % 2 === 0;
})
```

* Reduce.

- Performs some operations & reduces the array to a single value.
- It returns that single value.

* Dom : Part - 1 (tut 8)

* 3 Musketeers of Web Dev.
- HTML - CSS - JS.

* Window Object

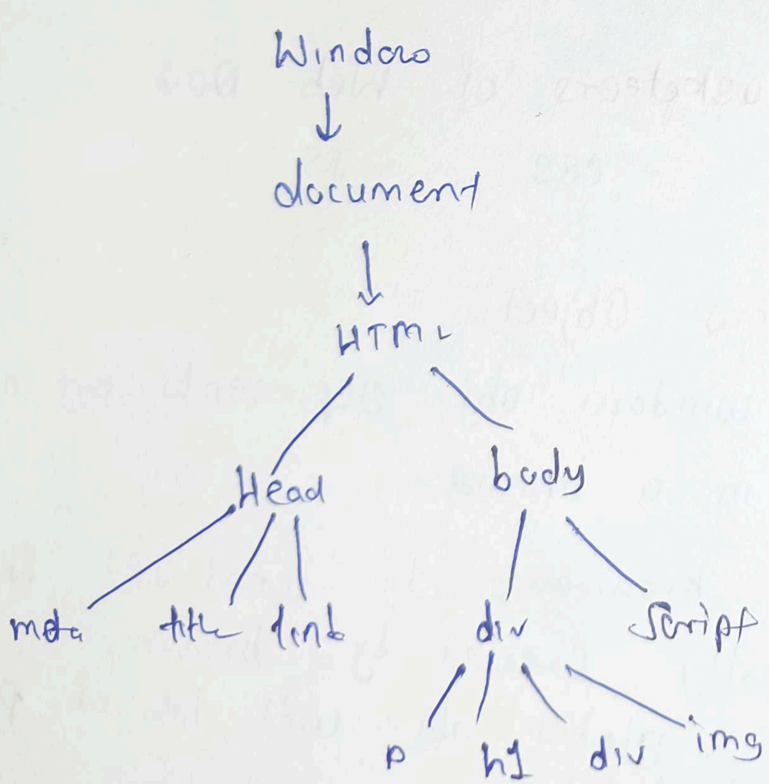
- The window obj. represents an open window in a browser.
- It is browser's obj. (not JS) & is automatically created by browser.
- It is a global obj. with lots of properties & methods

Console.log (window);

- Console.log ("hello");
- Window. console.log ("hellow");

* What is Dom ?

- Document Object Model
- It is programming interface for web docs.
- It ~~represents~~ the doc. as nodes & obj.
- DOM is basically the represent" of some HTML doc. but in ~~tree~~-like structure composed of obj.
- When a web page is loaded, the browser creates a DOM of page.



* DOM. Manipulation

- It allows developes to interact with & modify the structure, style & content of web pages

- Selecting with id.

```
document. getElement By Id ("myId");
```

- Selecting with class

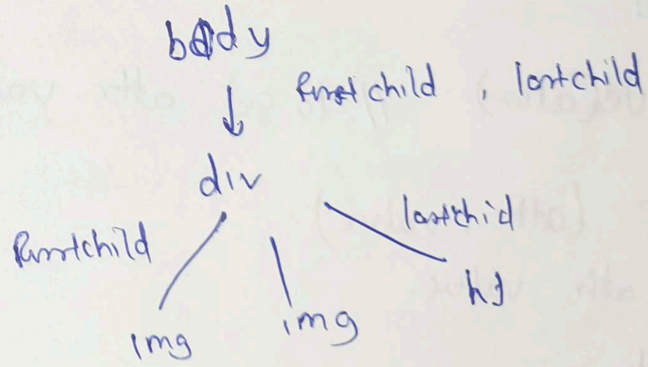
```
document. getElement By Class Name ("myclass");
```

- Selecting with tag.

```
document. getElement By Tag Name ("p");
```

- Query Selector.

```
document . querySelector ("myId / class / tag");
// returns 1st ele.

document . querySelector All ("myId / class / tag");
// Nodelist all elements
```



* Dom Manipulation Properties.

- tagName : returns tag for element nodes,
- innerText : returns the text content of the element & all its children.
- innerHTML : returns the plain text or HTML contents in elements
- textContent : returns textual content even for hidden elements

* Console.log ( ) ;
- display the object passed to it.

* Console.dir ( ) :
- display properties of that JS objects.

* DOM Part 2. (tut 8)

* Dom Manipulation.

* Attributes.

- getAttribute(attr) // to get attr value
- setAttribute (attr, value)
  // to set attr value.
- hasAttribute

* Style.

- node.style // to change style.

  eg. p.style.

* Insert Element.

Let el = document.createElement ("div");

- node.append (el) ; // adds at end of node (inside)
- node.prepend (el) ; // adds at start of node (inside)
- node.before (el) ; // adds before the node (outside)
- node.after (el) ; // adds after the node (outside)

* Delete Element

- node.remove ( ) ; // removes node.

* Events in JS. (See in all events) (tut 9)

- The change in the state of obj. is known as an Event.
- It provids a dynamic interface to webpages

eg.
- Mouse events (click, double click, etc)
- keyboard → (keypress, keyup)
- Form → (submit etc)
- Print event & many more.

* Event Handling

```
node.event = () => {
    // handle here
}
```

* Event Object

- It is a special obj. that has details about the event.
- All event handles have access to the event obj. properties & methods

```
node.event = (e) => {
    // handle here
}
```

// e.target, e.type, e.clientX, e.clientY.

* Event listeners

node.addEventListener(event, callback)

node.removeEventListener(event, callback);

* Event bubbling

- false : Register the event handler for the bubbling phase.
- True : Register the event handler for the capturing phase.

* Classes & Objects (tut 10)

- A JS obj is an entity having properties & methods
- JS objs have a special property called prototype.

syntax : --proto--

obj. creation :

```
const employee = {
    calcTax1 () {
      console.log("tax rate");
    },
    calcTax2 : function () {
      console.log("tax rate");
    }
};
```

* Classes in JS. (tut 11)

- Class is program code template for creating obj.
- These obj. will have some variable & Pn inside it.

```
class MyClass {
    constructor() {...}
    myMethod() {...}
}
let myobj = new MyClass();
```

* Constructor.

- It is automatically invoked by new obj
- It is used to initializes obj.

* Inheritance. (tut 12)

- It is used to derived prop. & methods from parent class to child class.

```
class Parent {
}

class child extends Parent {
}
```

* This keyword

- It refers to an obj. that is executing the current piece of code.

* Super Keyword. (tut 13)

- The super keyword is used to call the constructor of its parent class to access the parents properties & methods.

```
super (args)  // calls Parents constructor

super.parentMethod (args).
```

* Error Handling. (tut 14)

try - catch.

- try : It defines a code block to run (to try)
- catch : It defines a code block to handle any error.
- finally : It defines a code block to run regardless of result
- throw : It defines custom error.

```
try {
    .. normal code
} catch (err) {  // err is error obj.
    .. handling error
}
```

* Callbacks, Promises & Async-await. (tut 16)

* Sync in JS

- Synchronous : It means the code runs in a particular sequence of instructions given in program.
- Asynchronous :

- Asynchronous is non-blocking architecture, so the execution of one task isn't dependent on another.

* Callbacks.

- It is a fn that is passed as an argument to another fn. & is called after the main fn has finished its execution

* Callback Hell:

- Nested callbacks stacked below one another forming a pyramid structure.

* Promises.

- Promises is for "eventual" completion of task.
- It is an object in JS.
- It is a soln to callback hell.
- resolve & reject are callbacks provided by JS.

```
let promise = new Promise ( (resolve, reject) => {..} )
```

fn with 2 handlers

- A JS promise obj. can be:
  - Pending: the result is undefined
  - Resolved: the result is value (fulfilled)
  - Rejected: the result is an error object.

- Promise has state (pending, fulfilled) & some result.

* .then() & .catch()

```
promise.then ( (res) => {..} )  → resolve
promise.catch ( (err) => {..} )  → reject
```

* Async - await.

- The await keyword is used inside an async fn to pause its execut" & wait for a promise to resolve before continuing

```
async fn myfunc() {..}
```

* IIFE:

- Immediately Invoked fn Expression
- It is a fn that is called immediately as soon as it is defined.

* fetch API

- It provides an interface for fetching (sending / receiving) resources.
- It uses Request & Response obj.
- The fetch() method is used to fetch a resource (data)

```
let promise = fetch (url, [options])
```

* AJAX : It is Asynchronous JS & XML

* JSON : Javascript Obj, Notation

* JSon() method :
JSon() method returns a second Promise that resolves with result of parsing the Response body text as JSON.
(I/P is JSON, O/P is JS obj.).

* HTTP Request & Response Method

JavaScript Interview Questions for Freshers

1. What are the different data types present in javascript?

To know the type of a JavaScript variable, we can use the **typeof** operator.

1. Primitive types

String - It represents a series of characters and is written with quotes. A string can be represented using a single or a double quote.

Example :

```
var str = "Vivek Singh Bisht"; //using double quotes
var str2 = 'John Doe'; //using single quotes
```

- **Number** - It represents a number and can be written with or without decimals.

Example :

```
var x = 3; //without decimal
var y = 3.6; //with decimal
```

- **BigInt** - This data type is used to store numbers which are above the limitation of the Number data type. It can store large integers and is represented by adding "n" to an integer literal.

Example :

```
var bigInteger =  234567890123456789012345678901234567890;
```

- **Boolean** - It represents a logical entity and can have only two values : true or false. Booleans are generally used for conditional testing.

Example :

```
var a = 2;
var b =  3;
var c =  2;
(a == b) // returns false
(a == c) //returns true
```

- **Undefined** - When a variable is declared but not assigned, it has the value of undefined and it's type is also undefined.

Example :

```
var x; // value of x is undefined
var y = undefined; // we can also set the value of a variable as undefined
```

- **Null** - It represents a non-existent or a invalid value.

Example :

```
var z = null;
```

- **Symbol** - It is a new data type introduced in the ES6 version of javascript. It is used to store an anonymous and unique value.

Example :

```
var symbol1 = Symbol('symbol');
```

- typeof **of primitive types** :

```
typeof "John Doe" // Returns "string"
typeof 3.14 // Returns "number"
typeof true // Returns "boolean"
typeof 234567890123456789012345678901234567890n // Returns bigint
typeof undefined // Returns "undefined"
typeof null // Returns "object" (kind of a bug in JavaScript)
typeof Symbol('symbol') // Returns Symbol
```

2. Non-primitive types

- Primitive data types can store only a single value. To store multiple and complex values, non-primitive data types are used.
- Object - Used to store collection of data.
- Example:

```
// Collection of data in key-value pairs

var obj1 = {
    x:  43,
    y:  "Hello world!",
    z:  function(){
        return this.x;
```

```
typeof undefined // Returns "undefined"
typeof null // Returns "object" (kind of a bug in JavaScript)
typeof Symbol('symbol') // Returns Symbol
```

2. Non-primitive types

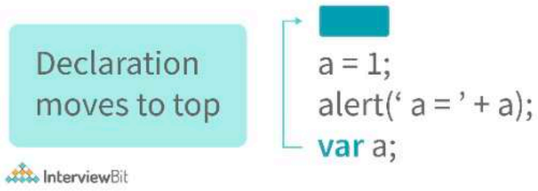
- Primitive data types can store only a single value. To store multiple and complex values, non-primitive data types are used.
- Object - Used to store collection of data.
- Example:

```
// Collection of data in key-value pairs

var obj1 = {
    x:  43,
    y:  "Hello world!",
    z: function(){
        return this.x;
    }
}

// Collection of data as an ordered list

var array1 = [5, "Hello", true, 4.1];
```

Note- It is important to remember that any data type that is not a primitive data type, is of Object type in javascript.

2. Explain Hoisting in javascript.

Hoisting is the default behaviour of javascript where all the variable and function declarations are moved on top.



This means that irrespective of where the variables and functions are declared, they are moved on top of the scope. The scope can be both local and global.

Example 1:

```
hoistedVariable = 3;
console.log(hoistedVariable); // outputs 3 even when the variable is declared after it is initialized
var hoistedVariable;
```

Example 2:

```
hoistedFunction();   // Outputs " Hello world! " even when the function is declared after calling

function hoistedFunction(){
   console.log(" Hello world! ");
}
```

Example 3:

```
// Hoisting takes place in the local scope as well
function doSomething(){
   x = 33;
   console.log(x);
   var x;
}

doSomething(); // Outputs 33 since the local variable "x" is hoisted inside the local scope
```

Note - Variable initializations are not hoisted, only variable declarations are hoisted:

```
var x;
console.log(x); // Outputs "undefined" since the initialization of "x" is not hoisted
x = 23;
```

Note - To avoid hoisting, you can run javascript in strict mode by using "use strict" on top of the code:

```
"use strict";
```

```
}
```

Example 3:

```
// Hoisting takes place in the local scope as well
function doSomething(){
  x = 33;
  console.log(x);
  var x;
}

doSomething(); // Outputs 33 since the local variable "x" is hoisted inside the local scope
```

Note - Variable initializations are not hoisted, only variable declarations are hoisted:

```
var x;
console.log(x);  // Outputs "undefined" since the initialization of "x" is not hoisted
x = 23;
```

Note - To avoid hoisting, you can run javascript in strict mode by using "use strict" on top of the code:

```
"use strict";
x = 23;  // Gives an error since 'x' is not declared
var x;
```

3. Why do we use the word "debugger" in javascript?

The debugger for the browser must be activated in order to debug the code. Built-in debuggers may be switched on and off, requiring the user to report faults. The remaining section of the code should stop execution before moving on to the next line while debugging.

4. Difference between " == " and " === " operators.

Both are comparison operators. The difference between both the operators is that "==" is used to compare values whereas, " === " is used to compare both values and types.

Example:

```
var x = 2;
var y = "2";
(x == y)   // Returns true since the value of both x and y is the same
(x === y) // Returns false since the typeof x is "number" and typeof y is "string"
```

5. Difference between var and let keyword in javascript.

Some differences are

1. From the very beginning, the 'var' keyword was used in JavaScript programming **whereas the keyword** 'let' was just added in 2015.
2. The keyword 'Var' has a function scope. Anywhere in the function, the variable specified using var is accessible but in 'let' the scope of a variable declared with the 'let' keyword is limited to the block in which it is declared. Let's start with a Block Scope.
3. In ECMAScript 2015, let and const are hoisted but not initialized. Referencing the variable in the block before the variable declaration results in a ReferenceError because the variable is in a "temporal dead zone" from the start of the block until the declaration is processed.

6. Explain Implicit Type Coercion in javascript.

Implicit type coercion in javascript is the automatic conversion of value from one data type to another. It takes place when the operands of an expression are of different data types.

- **String coercion**

String coercion takes place while using the ' + ' operator. When a number is added to a string, the number type is always converted to the string type.

Example 1:

```
var x = 3;
var y = "3";
x + y  // Returns "33"
```

Example 2:

```
var x = 24;
var y = "Hello";
x + y     // Returns "24Hello";
```

Note - ' + ' operator when used to add two numbers, outputs a
add two strings, outputs the cor

6. Explain Implicit Type Coercion in javascript.

Implicit type coercion in javascript is the automatic conversion of value from one data type to another. It takes place when the operands of an expression are of different data types.

- **String coercion**

String coercion takes place while using the ' + ' operator. When a number is added to a string, the number type is always converted to the string type.

Example 1:

```
var x = 3;
var y = "3";
x + y // Returns "33"
```

Example 2:

```
var x = 24;
var y = "Hello";
x + y    // Returns "24Hello";
```

Note - ' + ' operator when used to add two numbers, outputs a number. The same ' + ' operator when used to add two strings, outputs the concatenated string:

```
var name = "Vivek";
var surname = " Bisht";
name + surname       // Returns "Vivek Bisht"
```

Let's understand both the examples where we have added a number to a string,

When JavaScript sees that the operands of the expression x + y are of different types (one being a number type and the other being a string type), it converts the number type to the string type and then performs the operation. Since after conversion, both the variables are of string type, the ' + ' operator outputs the concatenated string "33" in the first example and "24Hello" in the second example.

Note - Type coercion also takes place when using the ' - ' operator, but the difference while using ' - ' operator is that, a string is converted to a number and then subtraction takes place.

```
var x = 3;
Var y = "3";
x - y      //Returns 0 since the variable y (string type) is converted to a number type
```

- **Boolean Coercion**

Boolean coercion takes place when using logical operators, ternary operators, if statements, and loop checks. To understand boolean coercion in if statements and operators, we need to understand truthy and falsy values.

Truthy values are those which will be converted (coerced) to **true**. Falsy values are those which will be converted to **false**.

All values except **false, 0, 0n, -0, "", null, undefined, and NaN** are truthy values.

If statements:

Example:

```
var x = 0;
var y = 23;

if(x) { console.log(x) }     // The code inside this block will not run since the value of x is 0(Falsy)

if(y) { console.log(y) }      // The code inside this block will run since the value of y is 23 (Truthy)
```

- **Logical operators:**

Logical operators in javascript, unlike operators in other programming languages, **do not return true or false. They always return one of the operands.**

OR (||) operator - If the first value is truthy, then the first value is returned. Otherwise, always the second value gets returned.

AND (&&) operator - If both the values are truthy, always the second value is returned. If the first value is falsy then the first value is returned or if the second value is falsy then the second value is returned.

Example:

```
var x = 220;
var y = "Hello";
var z = undefined;

x | | y      // Returns 220 since the first value is truthy

x | | z     // Returns 220 since the first value is truthy

x && y      // Returns "Hello" since both the values are truthy

y && z     // Returns undefined since the second value is falsy
```

Example:

```
var x = 220;
var y = "Hello";
var z = undefined;

x || y      // Returns 220 since the first value is truthy

x || z      // Returns 220 since the first value is truthy

x && y      // Returns "Hello" since both the values are truthy

y && z      // Returns undefined since the second value is falsy

if( x && y ){
  console.log("Code runs" ); // This block runs because x && y returns "Hello" (Truthy)
}

if( x || z ){
  console.log("Code runs"); // This block runs because x || y returns 220(Truthy)
}
```

- **Equality Coercion**

Equality coercion takes place when using ' == ' operator. As we have stated before

The ' == ' operator compares values and not types.

While the above statement is a simple way to explain == operator, it's not completely true

The reality is that while using the '==' operator, coercion takes place.

The '==' operator, converts both the operands to the same type and then compares them.

Example:

```
var a = 12;
var b = "12";
a == b // Returns true because both 'a' and 'b' are converted to the same type and then compared. Hence the operands are equal.
```

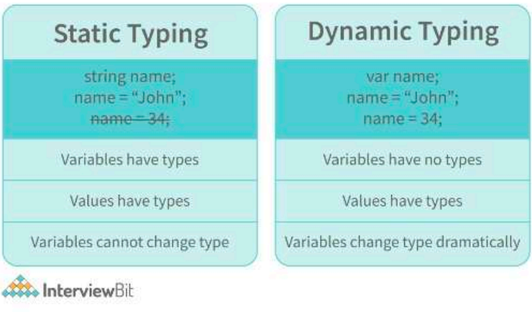
Coercion does not take place when using the '===' operator. Both operands are not converted to the same type in the case of '===' operator.

Example:

```
var a = 226;
var b = "226";

a === b // Returns false because coercion does not take place and the  operands are of different types. Hence they are not equal.
```

7. Is javascript a statically typed or a dynamically typed language?

JavaScript is a dynamically typed language. In a dynamically typed language, the type of a variable is checked during **run-time** in contrast to a statically typed language, where the type of a variable is checked during **compile-time.**



Since javascript is a loosely(dynamically) typed language, variables in JS are not associated with any type. A variable can hold the value of any data type.

For example, a variable that is assigned a number type can be converted to a string type:

```
var a = 23;
var a = "Hello World!";
```

8. What is NaN property in JavaScript?

NaN property represents the **"Not-a-Number"** value. It indicates a value tha

```
var a = 23;
var a = "Hello World!";
```

8. What is NaN property in JavaScript?

NaN property represents the **"Not-a-Number"** value. It indicates a value that is not a legal number.

typeof of NaN will return a **Number**.

To check if a value is NaN, we use the **isNaN()** function,

Note- isNaN() function converts the given value to a Number type, and then equates to NaN.

```
isNaN("Hello")   // Returns true
isNaN(345)    // Returns false
isNaN('1')   // Returns false, since '1' is converted to Number type which results in 0 ( a number)
isNaN(true)  // Returns false, since true converted to Number type results in 1 ( a number)
isNaN(false)  // Returns false
isNaN(undefined)  // Returns true
```

9. Explain passed by value and passed by reference.

In JavaScript, primitive data types are passed by value and non-primitive data types are passed by reference.

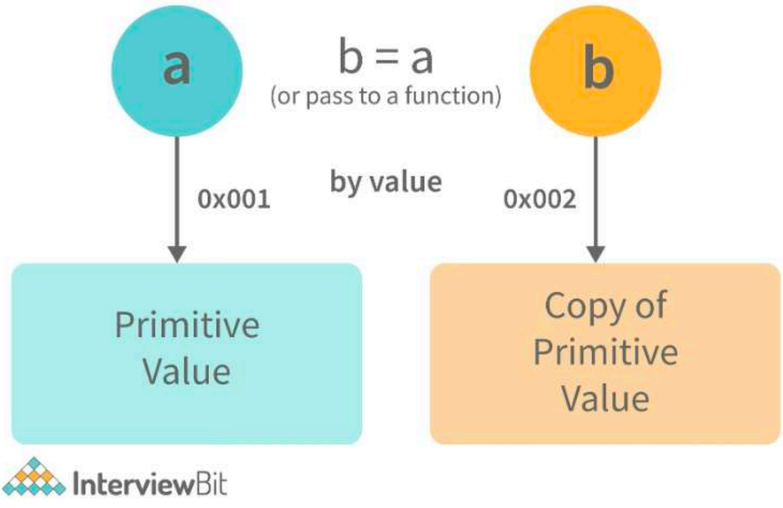
For understanding passed by value and passed by reference, we need to understand what happens when we create a variable and assign a value to it,

```
var x = 2;
```

In the above example, we created a variable x and assigned it a value of "2". In the background, the "=" (assign operator) allocates some space in the memory, stores the value "2" and returns the location of the allocated memory space. Therefore, the variable x in the above code points to the location of the memory space instead of pointing to the value 2 directly.

Assign operator behaves differently when dealing with primitive and non-primitive data types,

Assign operator dealing with primitive types:



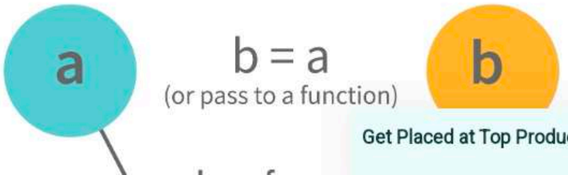

```
var y = 234;
var z = y;
```

In the above example, the assign operator knows that the value assigned to y is a primitive type (number type in this case), so when the second line code executes, where the value of y is assigned to z, the assign operator takes the value of y (234) and allocates a new space in the memory and returns the address. Therefore, variable z is not pointing to the location of variable y, instead, it is pointing to a new location in the memory.

```
var y = #8454;  // y pointing to address of the value 234

var z = y;

var z = #5411;  // z pointing to a completely new address of the value 234

// Changing the value of y
y = 23;
console.log(z);   // Returns 234, since z points to a new address in the memory so changes in y will not effect z
```

From the above example, we can see that primitive data types when passed to another variable, are passed by value. Instead of just assigning the same address to another variable, the value is passed and new space of memory is created.

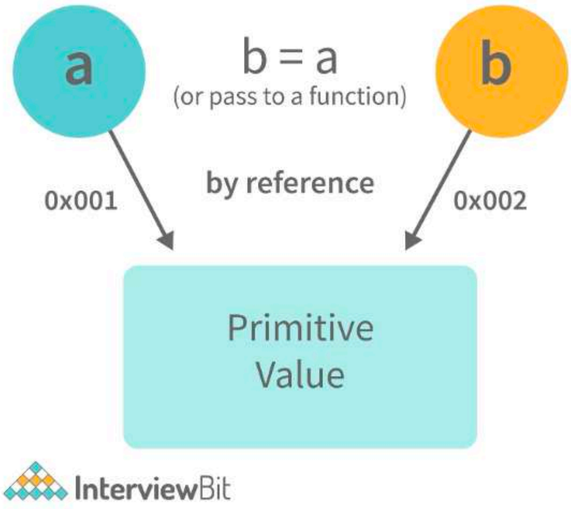
Assign operator dealing with non-primitive types:



```
// Changing the value of y
y = 23;
console.log(z);   // Returns 234, since z points to a new address in the memory so changes in y will not effect z
```

From the above example, we can see that primitive data types when passed to another variable, are passed by value. Instead of just assigning the same address to another variable, the value is passed and new space of memory is created.

Assign operator dealing with non-primitive types:




```
var obj = { name: "Vivek", surname: "Bisht" };
var obj2 = obj;
```

In the above example, the assign operator directly passes the location of the variable obj to the variable obj2. In other words, the reference of the variable obj is passed to the variable obj2.

```
var obj = #8711;   // obj pointing to address of { name: "Vivek", surname: "Bisht" }
var obj2 = obj;

var obj2 = #8711;  // obj2 pointing to the same address

// changing the value of obj1

obj.name = "Akki";
console.log(obj2);

// Returns {name:"Akki", surname:"Bisht"} since both the variables are pointing to the same address.
```

From the above example, we can see that while passing non-primitive data types, the assigned operator directly passes the address (reference).

Therefore, non-primitive data types are always **passed by reference.**

10. What is an Immediately Invoked Function in JavaScript?

An Immediately Invoked Function (known as IIFE and pronounced as IIFY) is a function that runs as soon as it is defined.

Syntax of IIFE :

```
(function(){
   // Do something;
})();
```

To understand IIFE, we need to understand the two sets of parentheses that are added while creating an IIFE :

The first set of parenthesis:

```
(function (){
    //Do something;
})
```

While executing javascript code, whenever the compiler sees the word “function”, it assumes that we are declaring a function in the code. Therefore, if we do not use the first set of parentheses, the compiler throws an error because it thinks we are declaring a function, and by the syntax of declaring a function, a function should always have a name.

```
function() {
   //Do something;
}
// Compiler gives an error since the syntax of declaring a function is wrong in the code above.
```

To remove this error, we add the first set of parenthesis that tells the compiler that the function is not a function declaration, instead, it’s a function expression.

The second set of parenthesis:

```
(function (){
   //Do something;
})();
```

code. Therefore, if we do not use the first set of parentheses, the compiler throws an error because it thinks we are declaring a function, and by the syntax of declaring a function, a function should always have a name.

```
function() {
  //Do something;
}
// Compiler gives an error since the syntax of declaring a function is wrong in the code above.
```

To remove this error, we add the first set of parenthesis that tells the compiler that the function is not a function declaration, instead, it's a function expression.

The second set of parenthesis:

```
(function (){
  //Do something;
})();
```

From the definition of an IIFE, we know that our code should run as soon as it is defined. A function runs only when it is invoked. If we do not invoke the function, the function declaration is returned:

```
(function (){
  // Do something;
})

// Returns the function declaration
```

Therefore to invoke the function, we use the second set of parenthesis.

11. What do you mean by strict mode in javascript and characteristics of javascript strict-mode?

In ECMAScript 5, a new feature called JavaScript Strict Mode allows you to write a code or a function in a "strict" operational environment. In most cases, this language is 'not particularly severe' when it comes to throwing errors. In 'Strict mode,' however, all forms of errors, including silent errors, will be thrown. As a result, debugging becomes a lot simpler. Thus programmer's chances of making an error are lowered.

Characteristics of strict mode in javascript

1. Duplicate arguments are not allowed by developers.
2. In strict mode, you won't be able to use the JavaScript keyword as a parameter or function name.
3. The 'use strict' keyword is used to define strict mode at the start of the script. Strict mode is supported by all browsers.
4. Engineers will not be allowed to create global variables in 'Strict Mode.

12. Explain Higher Order Functions in javascript.

Functions that operate on other functions, either by taking them as arguments or by returning them, are called higher-order functions.

Higher-order functions are a result of functions being **first-class citizens** in javascript.

Examples of higher-order functions:

```
function higherOrder(fn) {
  fn();
}

higherOrder(function() { console.log("Hello world") });

function higherOrder2() {
  return function() {
    return "Do something";
  }
}
var x = higherOrder2();
x()    // Returns "Do something"
```

13. Explain "this" keyword.

The "this" keyword refers to the object that the function is a property of.

The value of the "this" keyword will always depend on the object that is invoking the function.

Confused? Let's understand the above statements by examples:

```
function doSomething() {
  console.log(this);
}

doSomething();
```

What do you think the output of the above code will be?

Note - Observe the line where we are invoking the function.

Check the definition again:

The "this" keyword refers to the object that the function is a property of.

In the above code, the function is a property of which object?

```
}
var x = higherOrder2();
x()    // Returns "Do something"
```

13. Explain "this" keyword.

The "this" keyword refers to the object that the function is a property of.

The value of the "this" keyword will always depend on the object that is invoking the function.

Confused? Let's understand the above statements by examples:

```
function doSomething() {
  console.log(this);
}

doSomething();
```

What do you think the output of the above code will be?

Note - Observe the line where we are invoking the function.

Check the definition again:

The "this" keyword refers to the object that the function is a property of.

In the above code, the function is a property of which object?

Since the function is invoked in the global context, **the function is a property of the global object.**

Therefore, the output of the above code will be **the global object.** Since we ran the above code inside the browser, the global object is **the window object.**

Example 2:

```
var obj = {
    name:  "vivek",
    getName:  function(){
    console.log(this.name);
  }
}

obj.getName();
```

In the above code, at the time of invocation, the getName function is a property of the object **obj** , therefore, **this** keyword will refer to the object **obj**, and hence the output will be "vivek".

Example 3:

```
 var obj = {
    name:  "vivek",
    getName:  function(){
    console.log(this.name);
  }

}

var getName = obj.getName;

var obj2 = {name:"akshay", getName };
obj2.getName();
```

Can you guess the output here?

The output will be "akshay".

Although the getName function is declared inside the object **obj**, at the time of invocation, getName() is a property of **obj2**, therefore the "this" keyword will refer to **obj2**.

The silly way to understand the "**this**" keyword is, whenever the function is invoked, check the object before the **dot**. The value of **this** . keyword will always be the object before the **dot**.

If there is no object before the dot-like in example1, the value of this keyword will be the global object.

Example 4:

```
var obj1 = {
    address : "Mumbai,India",
    getAddress:  function(){
    console.log(this.address);
  }
}

var getAddress = obj1.getAddress;
var obj2 = {name:"akshay"};
obj2.getAddress();
```

Can you guess the output?

The output will be an error.

Example 4:

```
var obj1 = {
    address : "Mumbai,India",
    getAddress: function(){
    console.log(this.address);
  }
}

var getAddress = obj1.getAddress;
var obj2 = {name:"akshay"};
obj2.getAddress();
```

Can you guess the output?

The output will be an error.

Although in the code above, this keyword refers to the object **obj2**, obj2 does not have the property "address"', hence the getAddress function throws an error.

14. What do you mean by Self Invoking Functions?

Without being requested, a self-invoking expression is automatically invoked (initiated). If a function expression is followed by (), it will execute automatically. A function declaration cannot be invoked by itself.

Normally, we declare a function and call it, however, anonymous functions may be used to run a function automatically when it is described and will not be called again. And there is no name for these kinds of functions.

15. Explain call(), apply() and, bind() methods.

1. call():

- It's a predefined method in javascript.
- This method invokes a method (function) by specifying the owner object.
- Example 1:

```
function sayHello(){
  return "Hello " + this.name;
}

var obj = {name: "Sandy"};

sayHello.call(obj);

// Returns "Hello Sandy"
```

- call() method allows an object to use the method (function) of another object.
- Example 2:

```
var person = {
  age: 23,
  getAge: function(){
    return this.age;
  }
}
var person2 = {age: 54};
person.getAge.call(person2);
// Returns 54
```

- call() accepts arguments:

```
function saySomething(message){
  return this.name + "is" + message;
}
var person4 = {name: "John"};
saySomething.call(person4, "awesome");
// Returns "John is awesome"
```

apply()

The apply method is similar to the call() method. The only difference is that,

call() method takes arguments separately whereas, apply() method takes arguments as an array.

```
function saySomething(message){
  return this.name + "is" + message;
}
var person4 = {name: "John"};
saySomething.apply(person4, ["awesome"]);
```

2. bind():

- This method returns a new function, where the value of **"this"** keyword will be bound to the owner object, which is provided as a parameter.
- Example with arguments:

```
var bikeDetails = {
    displayDetails: function(registrationNumber,brandName){
    return this.name+ " "+ "bike details: "+ registrationNumber +
```

apply()

The apply method is similar to the call() method. The only difference is that,

call() method takes arguments separately whereas, apply() method takes arguments as an array.

```
function saySomething(message){
  return this.name + "is" + message;
}
var person4 = {name:  "John"};
saySomething.apply(person4, ["awesome"]);
```

2. bind():

- This method returns a new function, where the value of **"this"** keyword will be bound to the owner object, which is provided as a parameter.
- Example with arguments:

```
var bikeDetails = {
    displayDetails:  function(registrationNumber,brandName){
    return this.name+ ","+ "bike details: "+ registrationNumber + "," + brandName;
  }
}

var person1 = {name:  "Vivek"};

var detailsOfPerson1 = bikeDetails.displayDetails.bind(person1, "TS0122", "Bullet");

// Binds the displayDetails function to the person1 object


detailsOfPerson1();
//Returns Vivek, bike details: TS0122, Bullet
```

16. What is the difference between exec () and test () methods in javascript?

- **test ()** and **exec ()** are RegExp expression methods used in javascript.
- We'll use **exec ()** to search a string for a specific pattern, and if it finds it, it'll return the pattern directly; else, it'll return an 'empty' result.
- We will use a **test ()** to find a string for a specific pattern. It will return the Boolean value 'true' on finding the given text otherwise, it will return 'false'.

17. What is currying in JavaScript?

Currying is an advanced technique to transform a function of arguments n, to n functions of one or fewer arguments.

Example of a curried function:

```
function add (a) {
  return function(b){
    return a + b;
  }
}

add(3)(4)
```

For Example, if we have a function **f(a,b)**, then the function after currying, will be transformed to **f(a)(b).**

By using the currying technique, we do not change the functionality of a function, we just change the way it is invoked.

Let's see currying in action:

```
function multiply(a,b){
  return a*b;
}

function currying(fn){
  return function(a){
    return function(b){
      return fn(a,b);
    }
  }
}

var curriedMultiply = currying(multiply);

multiply(4, 3);  // Returns 12

curriedMultiply(4)(3);  // Also returns 12
```

As one can see in the code above, we have transformed the function **multiply(a,b)** to a function **curriedMultiply** , which takes in one parameter at a time.

18. What are some advantages of using External JavaScript?

External JavaScript is the JavaScript Code (script) written in a separate file

As one can see in the code above, we have transformed the function **multiply(a,b)** to a function **curriedMultiply** , which takes in one parameter at a time.

18. What are some advantages of using External JavaScript?

External JavaScript is the JavaScript Code (script) written in a separate file with the extension.js, and then we link that file inside the <head> or <body> element of the HTML file where the code is to be placed.

Some advantages of external javascript are

1. It allows web designers and developers to collaborate on HTML and javascript files.
2. We can reuse the code.
3. Code readability is simple in external javascript.

19. Explain Scope and Scope Chain in javascript.

Scope in JS determines the accessibility of variables and functions at various parts of one's code.

In general terms, the scope will let us know at a given part of code, what are variables and functions we can or cannot access.

There are three types of scopes in JS:

- Global Scope
- Local or Function Scope
- Block Scope

Global Scope: Variables or functions declared in the global namespace have global scope, which means all the variables and functions having global scope can be accessed from anywhere inside the code.

```
var globalVariable = "Hello world";

function sendMessage(){
  return globalVariable; // can access globalVariable since it's written in global space
}
function sendMessage2(){
  return sendMessage(); // Can access sendMessage function since it's written in global space
}
sendMessage2();   // Returns "Hello world"
```

Function Scope: Any variables or functions declared inside a function have local/function scope, which means that all the variables and functions declared inside a function, can be accessed from within the function and not outside of it.

```
function awesomeFunction(){
  var a = 2;

  var multiplyBy2 = function(){
    console.log(a*2); // Can access variable "a" since a and multiplyBy2 both are written inside the same function
  }
}
console.log(a); // Throws reference error since a is written in local scope and cannot be accessed outside

multiplyBy2(); // Throws reference error since multiplyBy2 is written in local scope
```

Block Scope: Block scope is related to the variables declared using let and const. Variables declared with var do not have block scope. Block scope tells us that any variable declared inside a block {}, can be accessed only inside that block and cannot be accessed outside of it.

```
{
  let x = 45;
}

console.log(x); // Gives reference error since x cannot be accessed outside of the block

for(let i=0; i<2; i++){
  // do something
}

console.log(i); // Gives reference error since i cannot be accessed outside of the for loop block
```

Scope Chain: JavaScript engine also uses Scope to find variables. Let's understand that using an example:

```
var y = 24;

function favFunction(){
  var x = 667;
  var anotherFavFunction = function(){
    console.log(x); // Does not find x inside anotherFavFunction, so looks for variable inside favFunction, outputs 667
  }

  var yetAnotherFavFunction = function(){
    console.log(y); // Does not find y inside yetAnotherFavFunction, so looks for variable inside favFunction and does not find it
  }

  anotherFavFunction();
  yetAnotherFavFunction();
}
```

```
}

console.log(i);  // Gives reference error since i cannot be accessed outside of the for loop block
```

Scope Chain: JavaScript engine also uses Scope to find variables. Let's understand that using an example:

```
var y = 24;

function favFunction(){
  var x = 667;
  var anotherFavFunction = function(){
    console.log(x);  // Does not find x inside anotherFavFunction, so looks for variable inside favFunction, outputs 667
  }

  var yetAnotherFavFunction = function(){
    console.log(y);  // Does not find y inside yetAnotherFavFunction, so looks for variable inside favFunction and does not find it
  }

  anotherFavFunction();
  yetAnotherFavFunction();
}
favFunction();
```

As you can see in the code above, if the javascript engine does not find the variable in local scope, it tries to check for the variable in the outer scope. If the variable does not exist in the outer scope, it tries to find the variable in the global scope.

If the variable is not found in the global space as well, a reference error is thrown.

20. Explain Closures in JavaScript.

Closures are an ability of a function to remember the variables and functions that are declared in its outer scope.

```
var Person = function(pName){
  var name = pName;

  this.getName = function(){
    return name;
  }
}

var person = new Person("Neelesh");
console.log(person.getName());
```

Let's understand closures by example:

```
function randomFunc(){
  var obj1 = {name:"Vivian", age:45};

  return function(){
    console.log(obj1.name + "is"+ "awesome");  // Has access to obj1 even when the randomFunc function is executed

  }
}

var initialiseClosure = randomFunc();  // Returns a function

initialiseClosure();
```

Let's understand the code above,

The function randomFunc() gets executed and returns a function when we assign it to a variable:

```
var initialiseClosure = randomFunc();
```

The returned function is then executed when we invoke initialiseClosure:

```
initialiseClosure();
```

The line of code above outputs "Vivian is awesome" and this is possible because of closure.

```
console.log(obj1.name + "is"+ "awesome");
```

When the function randomFunc() runs, it seems that the returning function is using the variable obj1 inside it:

Therefore randomFunc(), instead of destroying the value of obj1 after execution, **saves the value in the memory for further reference.** This is the reason why the returning function is able to use the variable declared in the outer scope even after the function is already executed.

This ability of a function to store a variable for further reference even after it is executed is called Closure.

21. Mention some advantages of javascript.

There are many advantages of javascript. Some of them are

1. Javascript is executed on the client-side as well as server-side also. There are a variety of Frontend Frameworks that you may study and utilize. However, if you want to use JavaScript on the backend, you'll need to learn NodeJS. It is currently the only JavaScript framework that may be used on the backend.
2. Javascript is a simple language to learn.

Therefore randomFunc(), instead of destroying the value of obj1 after execution, **saves the value in the memory for further reference.** This is the reason why the returning function is able to use the variable declared in the outer scope even after the function is already executed.

This ability of a function to store a variable for further reference even after it is executed is called Closure.

21. Mention some advantages of javascript.

There are many advantages of javascript. Some of them are

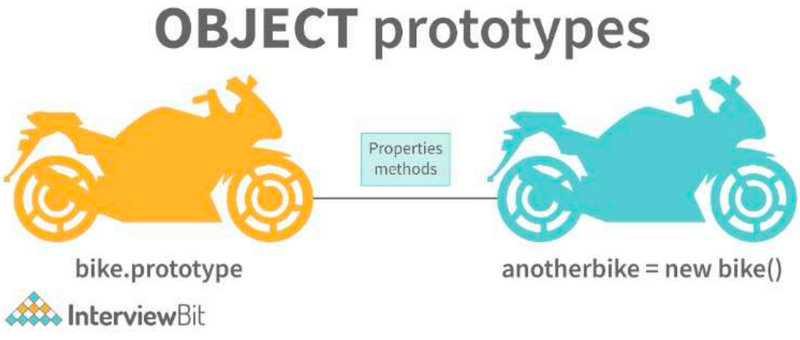
1. Javascript is executed on the client-side as well as server-side also. There are a variety of Frontend Frameworks that you may study and utilize. However, if you want to use JavaScript on the backend, you'll need to learn NodeJS. It is currently the only JavaScript framework that may be used on the backend.
2. Javascript is a simple language to learn.
3. Web pages now have more functionality because of Javascript.
4. To the end-user, Javascript is quite quick.

22. What are object prototypes?

All javascript objects inherit properties from a prototype. For example,

- Date objects inherit properties from the Date prototype
- Math objects inherit properties from the Math prototype
- Array objects inherit properties from the Array prototype.
- On top of the chain is **Object.prototype.** Every prototype inherits properties and methods from the Object.prototype.
- **A prototype is a blueprint of an object. The prototype** allows us to use properties and methods on an object even if the properties and methods do not exist on the current object.

Let's see prototypes help us use methods and properties:




```
var arr = [];
arr.push(2);

console.log(arr); // Outputs [2]
```

In the code above, as one can see, we have not defined any property or method called push on the array "arr" but the javascript engine does not throw an error.

The reason is the use of prototypes. As we discussed before, Array objects inherit properties from the Array prototype.

The javascript engine sees that the method push does not exist on the current array object and therefore, looks for the method push inside the Array prototype and it finds the method.

Whenever the property or method is not found on the current object, the javascript engine will always try to look in its prototype and if it still does not exist, it looks inside the prototype's prototype and so on.

23. What are callbacks?

A callback is a function that will be executed after another function gets executed. In javascript, functions are treated as first-class citizens, they can be used as an argument of another function, can be returned by another function, and can be used as a property of an object.

Functions that are used as an argument to another function are called callback functions. Example:

```
function divideByHalf(sum){
  console.log(Math.floor(sum / 2));
}

function multiplyBy2(sum){
  console.log(sum * 2);
}

function operationOnSum(num1,num2,operation){
  var sum = num1 + num2;
  operation(sum);
}
```

Whenever the property or method is not found on the current object, the javascript engine will always try to look in its prototype and if it still does not exist, it looks inside the prototype's prototype and so on.

23. What are callbacks?

A callback is a function that will be executed after another function gets executed. In javascript, functions are treated as first-class citizens, they can be used as an argument of another function, can be returned by another function, and can be used as a property of an object.

Functions that are used as an argument to another function are called callback functions. Example:

```
function divideByHalf(sum){
  console.log(Math.floor(sum / 2));
}

function multiplyBy2(sum){
  console.log(sum * 2);
}

function operationOnSum(num1,num2,operation){
  var sum = num1 + num2;
  operation(sum);
}

operationOnSum(3, 3, divideByHalf); // Outputs 3

operationOnSum(5, 5, multiplyBy2); // Outputs 20
```

- In the code above, we are performing mathematical operations on the sum of two numbers. The operationOnSum function takes 3 arguments, the first number, the second number, and the operation that is to be performed on their sum (callback).
- Both divideByHalf and multiplyBy2 functions are used as callback functions in the code above.
- These callback functions will be executed only after the function operationOnSum is executed.
- Therefore, a callback is a function that will be executed after another function gets executed.

24. What are the types of errors in javascript?

There are two types of errors in javascript.

1. **Syntax error**: Syntax errors are mistakes or spelling problems in the code that cause the program to not execute at all or to stop running halfway through. Error messages are usually supplied as well.
2. **Logical error**: Reasoning mistakes occur when the syntax is proper but the logic or program is incorrect. The application executes without problems in this case. However, the output findings are inaccurate. These are sometimes more difficult to correct than syntax issues since these applications do not display error signals for logic faults.

25. What is memoization?

Memoization is a form of caching where the return value of a function is cached based on its parameters. If the parameter of that function is not changed, the cached version of the function is returned.
Let's understand memoization, by converting a simple function to a memoized function:

> Note- Memoization is used for expensive function calls but in the following example, we are considering a simple function for understanding the concept of memoization better.

Consider the following function:

```
function addTo256(num){
  return num + 256;
}
addTo256(20); // Returns 276
addTo256(40); // Returns 296
addTo256(20); // Returns 276
```

In the code above, we have written a function that adds the parameter to 256 and returns it.

When we are calling the function addTo256 again with the same parameter ("20" in the case above), we are computing the result again for the same parameter.

Computing the result with the same parameter, again and again, is not a big deal in the above case, but imagine if the function does some heavy-duty work, then, computing the result again and again with the same parameter will lead to wastage of time.

This is where memoization comes in, by using memoization we can store(cache) the computed results based on the parameters. If the same parameter is used again while invoking the function, instead of computing the result, we directly return the stored (cached) value.

Let's convert the above function addTo256, to a memoized function:

```
function memoizedAddTo256(){
  var cache = {};

  return function(num){
    if(num in cache){
      console.log("cached value");
      return cache[num]
    }
    else{
      cache[num] = num + 256;
```

Let's convert the above function addTo256, to a memoized function:

```
function memoizedAddTo256(){
  var cache = {};

  return function(num){
    if(num in cache){
      console.log("cached value");
      return cache[num]
    }
    else{
      cache[num] = num + 256;
      return cache[num];
    }
  }
}
var memoizedFunc = memoizedAddTo256();

memoizedFunc(20); // Normal return
memoizedFunc(20); // Cached return
```

In the code above, if we run the memoizedFunc function with the same parameter, instead of computing the result again, it returns the cached result.

> Note- Although using memoization saves time, it results in larger consumption of memory since we are storing all the computed results.

26. What is recursion in a programming language?

Recursion is a technique to iterate over an operation by having a function call itself repeatedly until it arrives at a result.

```
function add(number) {
  if (number <= 0) {
    return 0;
  } else {
    return number + add(number - 1);
  }
}
add(3) => 3 + add(2)
          3 + 2 + add(1)
          3 + 2 + 1 + add(0)
          3 + 2 + 1 + 0 = 6
```

Example of a recursive function:

The following function calculates the sum of all the elements in an array by using recursion:

```
function computeSum(arr){
  if(arr.length === 1){
    return arr[0];
  }
  else{
    return arr.pop() + computeSum(arr);
  }
}
computeSum([7, 8, 9, 99]); // Returns 123
```


27. What is the use of a constructor function in javascript?

Constructor functions are used to create objects in javascript.

When do we use constructor functions?

If we want to create multiple objects having similar properties and methods, constructor functions are used.

Note- The name of a constructor function should always be written in Pascal Notation: every word should start with a capital letter.

Example:

```
function Person(name,age,gender){
  this.name = name;
  this.age = age;
  this.gender = gender;
}

var person1 = new Person("Vivek", 76, "male");
console.log(person1);
```

Recursion Medium 13.46 Mins Solve

27. What is the use of a constructor function in javascript?

Constructor functions are used to create objects in javascript.

When do we use constructor functions?

If we want to create multiple objects having similar properties and methods, constructor functions are used.

Note- The name of a constructor function should always be written in Pascal Notation: every word should start with a capital letter.

Example:

```
function Person(name,age,gender){
  this.name = name;
  this.age = age;
  this.gender = gender;
}


var person1 = new Person("Vivek", 76, "male");
console.log(person1);

var person2 = new Person("Courtney", 34, "female");
console.log(person2);
```

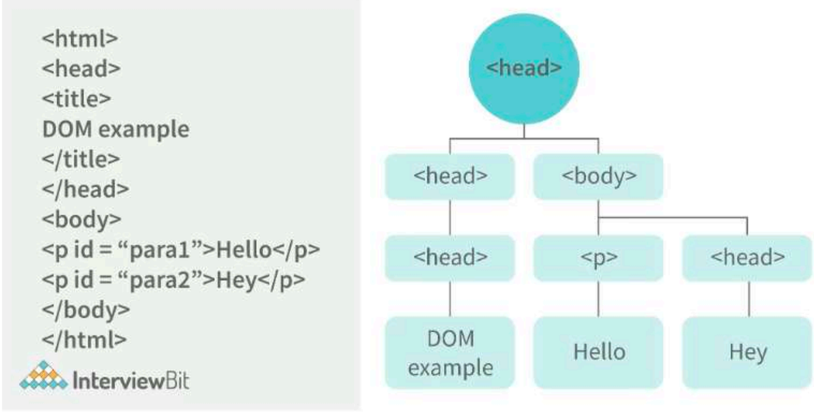
In the code above, we have created a constructor function named Person. Whenever we want to create a new object of the type Person, We need to create it using the new keyword:

```
var person3 = new Person("Lilly", 17, "female");
```

The above line of code will create a new object of the type Person. Constructor functions allow us to group similar objects.

28. What is DOM?

- DOM stands for Document Object Model. DOM is a programming interface for HTML and XML documents.
- When the browser tries to render an HTML document, it creates an object based on the HTML document called DOM. Using this DOM, we can manipulate or change various elements inside the HTML document.
- Example of how HTML code gets converted to DOM:



29. Which method is used to retrieve a character from a certain index?

The charAt() function of the JavaScript string finds a char element at the supplied index. The index number begins at 0 and continues up to n-1, Here n is the string length. The index value must be positive, higher than, or the same as the string length.

30. What do you mean by BOM?

Browser Object Model is known as BOM. It allows users to interact with the browser. A browser's initial object is a window. As a result, you may call all of the window's functions directly or by referencing the window. The document, history, screen, navigator, location, and other attributes are available in the window object.

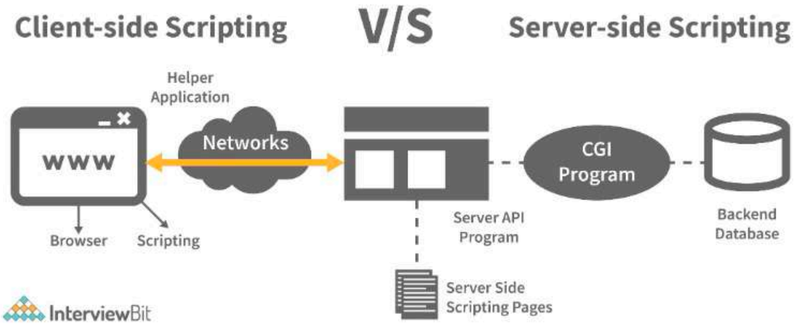
31. What is the distinction between client-side and server-side JavaScript?

Client-side JavaScript is made up of two parts, a fundamental language and predefined objects for performing JavaScript in a browser. JavaScript for the client is automatically included in the HTML pages. At runtime, the browser understands this script.

location, and other attributes are available in the window object.

31. What is the distinction between client-side and server-side JavaScript?

Client-side JavaScript is made up of two parts, a fundamental language and predefined objects for performing JavaScript in a browser. JavaScript for the client is automatically included in the HTML pages. At runtime, the browser understands this script.



Server-side JavaScript, involves the execution of JavaScript code on a server in response to client requests. It handles these requests and delivers the relevant response to the client, which may include client-side JavaScript for subsequent execution within the browser.

JavaScript Interview Questions for Experienced

32. What are arrow functions?

Arrow functions were introduced in the ES6 version of javascript. They provide us with a new and shorter syntax for declaring functions. Arrow functions can only be used as a function expression.

Let's compare the normal function declaration and the arrow function declaration in detail:

```
// Traditional Function Expression
var add = function(a,b){
  return a + b;
}

// Arrow Function Expression
var arrowAdd = (a,b) => a + b;
```

Arrow functions are declared without the function keyword. If there is only one returning expression then we don't need to use the return keyword as well in an arrow function as shown in the example above. Also, for functions having just one line of code, curly braces {} can be omitted.

```
// Traditional function expression
var multiplyBy2 = function(num){
  return num * 2;
}
// Arrow function expression
var arrowMultiplyBy2 = num => num * 2;
```

If the function takes in only one argument, then the parenthesis () around the parameter can be omitted as shown in the code above.

```
var obj1 = {
  valueOfThis: function(){
    return this;
  }
}
var obj2 = {
  valueOfThis: ()=>{
    return this;
  }
}

obj1.valueOfThis(); // Will return the object obj1
obj2.valueOfThis(); // Will return window/global object
```

The biggest difference between the traditional function expression and the arrow function is the handling of **this** keyword. By general definition, **this** keyword always refers to the object that is calling the function. As you can see in the code above, **obj1.valueOfThis()** returns obj1 since **this** keyword refers to the object calling the function.

In the arrow functions, there is no binding of **this** keyword. This keyword inside an arrow function does not refer to the object calling it. It rather inherits its value from the parent scope which is the window object in this case. Therefore, in the code above, **obj2.valueOfThis()** returns the window object.

```
obj2.valueOfThis(); // Will return window/global object
```

The biggest difference between the traditional function expression and the arrow function is the handling of **this** keyword. By general definition, **this** keyword always refers to the object that is calling the function. As you can see in the code above, **obj1.valueOfThis()** returns obj1 since **this** keyword refers to the object calling the function.

In the arrow functions, there is no binding of **this** keyword. This keyword inside an arrow function does not refer to the object calling it. It rather inherits its value from the parent scope which is the window object in this case. Therefore, in the code above, **obj2.valueOfThis()** returns the window object.

33. What do mean by prototype design pattern?

The Prototype Pattern produces different objects, but instead of returning uninitialized objects, it produces objects that have values replicated from a template – or sample – object. Also known as the Properties pattern, the Prototype pattern is used to create prototypes.

The introduction of business objects with parameters that match the database's default settings is a good example of where the Prototype pattern comes in handy. The default settings for a newly generated business object are stored in the prototype object.

The Prototype pattern is hardly used in traditional languages, however, it is used in the development of new objects and templates in JavaScript, which is a prototypal language.

34. Differences between declaring variables using var, let and const.

Before the ES6 version of javascript, only the keyword var was used to declare variables. With the ES6 Version, keywords let and const were introduced to declare variables.

keyword	const	let	var
global scope	no	no	yes
function scope	yes	yes	yes
block scope	yes	yes	no
can be reassigned	no	yes	yes

Let's understand the differences with examples:

```
var variable1 = 23;

let variable2 = 89;

function catchValues(){
  console.log(variable1);
  console.log(variable2);

// Both the variables can be accessed anywhere since they are declared in the global scope
}

window.variable1; // Returns the value 23

window.variable2; // Returns undefined
```

- The variables declared with the let keyword in the global scope behave just like the variable declared with the var keyword in the global scope.
- Variables declared in the global scope with var and let keywords can be accessed from anywhere in the code.
- But, there is one difference! Variables that are declared with the var keyword in the global scope are added to the window/global object. Therefore, they can be accessed using window.variableName.
  Whereas, the variables declared with the let keyword are not added to the global object, therefore, trying to access such variables using window.variableName results in an error.

var vs let in functional scope

```
function varVsLetFunction(){
  let awesomeCar1 = "Audi";
  var awesomeCar2 = "Mercedes";
}

console.log(awesomeCar1); // Throws an error
console.log(awesomeCar2); // Throws an error
```

Variables are declared in a functional/local scope using **var** and **let** keywords behave exactly the same, meaning, they cannot be accessed from outside of the scope.

```
{
  var variable3 = [1, 2, 3, 4];
}

console.log(variable3); // Outputs [1,2,3,4]

{
```

variables using window.variableName results in an error.

var vs let in functional scope

```
function varVsLetFunction(){
  let awesomeCar1 = "Audi";
  var awesomeCar2 = "Mercedes";
}

console.log(awesomeCar1); // Throws an error
console.log(awesomeCar2); // Throws an error
```

Variables are declared in a functional/local scope using **var** and **let** keywords behave exactly the same, meaning, they cannot be accessed from outside of the scope.

```
{
  var variable3 = [1, 2, 3, 4];
}

console.log(variable3); // Outputs [1,2,3,4]

{
  let variable4 = [6, 55, -1, 2];
}

console.log(variable4); // Throws error

for(let i = 0; i < 2; i++){
  //Do something
}

console.log(i); // Throws error

for(var j = 0; j < 2; i++){
  // Do something
}

console.log(j) // Outputs 2
```

- In javascript, a block means the code written inside the curly braces **{}**.
- Variables declared with **var** keyword do not have block scope. It means a variable declared in block scope **{}** with the **var** keyword is the same as declaring the variable in the global scope.
- Variables declared with **let** keyword inside the block scope cannot be accessed from outside of the block.

Const keyword

- Variables with the **const** keyword behave exactly like a variable declared with the let keyword with only one difference, **any variable declared with the const keyword cannot be reassigned.**
- Example:

```
const x = {name:"Vivek"};

x = {address: "India"}; // Throws an error

x.name = "Nikhil"; // No error is thrown

const y = 23;

y = 44; // Throws an error
```

In the code above, although we can change the value of a property inside the variable declared with **const** keyword, we cannot completely reassign the variable itself.

35. What is the rest parameter and spread operator?

Both rest parameter and spread operator were introduced in the ES6 version of javascript.

Rest parameter (...):

- It provides an improved way of handling the parameters of a function.
- Using the rest parameter syntax, we can create functions that can take a variable number of arguments.
- Any number of arguments will be converted into an array using the rest parameter.
- It also helps in extracting all or some parts of the arguments.
- Rest parameters can be used by applying three dots (...) before the parameters.

```
function extractingArgs(...args){
  return args[1];
}

// extractingArgs(8,9,1); // Returns 9

function addAllArgs(...args){
  let sumOfArgs = 0;
  let i = 0;
  while(i < args.length){
    sumOfArgs += args[i];
    i++;
```

- It also helps in extracting all or some parts of the arguments.
- Rest parameters can be used by applying three dots (...) before the parameters.

```
function extractingArgs(...args){
  return args[1];
}

// extractingArgs(8,9,1); // Returns 9

function addAllArgs(...args){
  let sumOfArgs = 0;
  let i = 0;
  while(i < args.length){
    sumOfArgs += args[i];
    i++;
  }
  return sumOfArgs;
}

addAllArgs(6, 5, 7, 99); // Returns 117
addAllArgs(1, 3, 4); // Returns 8
```

**Note- Rest parameter should always be used at the last parameter of a function:

```
// Incorrect way to use rest parameter
function randomFunc(a,...args,c){
//Do something
}

// Correct way to use rest parameter
function randomFunc2(a,b,...args){
//Do something
}
```

- **Spread operator (...):** Although the syntax of the spread operator is exactly the same as the rest parameter, the spread operator is used to spreading an array, and object literals. We also use spread operators where one or more arguments are expected in a function call.

```
function addFourNumbers(num1,num2,num3,num4){
  return num1 + num2 + num3 + num4;
}

let fourNumbers = [5, 6, 7, 8];


addFourNumbers(...fourNumbers);
// Spreads [5,6,7,8] as 5,6,7,8

let array1 = [3, 4, 5, 6];
let clonedArray1 = [...array1];
// Spreads the array into 3,4,5,6
console.log(clonedArray1); // Outputs [3,4,5,6]


let obj1 = {x:'Hello', y:'Bye'};
let clonedObj1 = {...obj1}; // Spreads and clones obj1
console.log(obj1);

let obj2 = {z:'Yes', a:'No'};
let mergedObj = {...obj1, ...obj2}; // Spreads both the objects and merges it
console.log(mergedObj);
// Outputs {x:'Hello', y:'Bye',z:'Yes',a:'No'};
```

***Note- Key differences between rest parameter and spread operator:

- Rest parameter is used to take a variable number of arguments and turns them into an array while the spread operator takes an array or an object and spreads it
- Rest parameter is used in function declaration whereas the spread operator is used in function calls.

36. In JavaScript, how many different methods can you make an object?

In JavaScript, there are several ways to declare or construct an object.

1. Object.
2. using Class.
3. create Method.
4. Object Literals.
5. using Function.
6. Object Constructor.

37. What is the use of promises in javascript?

Promises are used to handle asynchronous operations in javascript.

Before promises, callbacks were used to handle asynchronous operations. multiple callbacks to handle asynchronous code can lead to unmanageable

3. create Method.
4. Object Literals.
5. using Function.
6. Object Constructor.

37. What is the use of promises in javascript?

Promises are used to handle asynchronous operations in javascript.

Before promises, callbacks were used to handle asynchronous operations. But due to the limited functionality of callbacks, using multiple callbacks to handle asynchronous code can lead to unmanageable code.

Promise object has four states -

- Pending - Initial state of promise. This state represents that the promise has neither been fulfilled nor been rejected, it is in the pending state.
- Fulfilled - This state represents that the promise has been fulfilled, meaning the async operation is completed.
- Rejected - This state represents that the promise has been rejected for some reason, meaning the async operation has failed.
- Settled - This state represents that the promise has been either rejected or fulfilled.

A promise is created using the **Promise** constructor which takes in a callback function with two parameters, **resolve** and **reject** respectively.

resolve is a function that will be called when the async operation has been successfully completed.

reject is a function that will be called, when the async operation fails or if some error occurs.

Example of a promise:

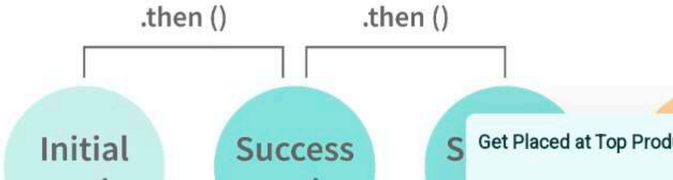
Promises are used to handle asynchronous operations like server requests, for ease of understanding, we are using an operation to calculate the sum of three elements.

In the function below, we are returning a promise inside a function:

```
function sumOfThreeElements(...elements){
  return new Promise((resolve,reject)=>{
    if(elements.length > 3 ){
      reject("Only three elements or less are allowed");
    }
    else{
      let sum = 0;
      let i = 0;
      while(i < elements.length){
        sum += elements[i];
        i++;
      }
      resolve("Sum has been calculated: "+sum);
    }
  })
}
```

In the code above, we are calculating the sum of three elements, if the length of the elements array is more than 3, a promise is rejected, or else the promise is resolved and the sum is returned.

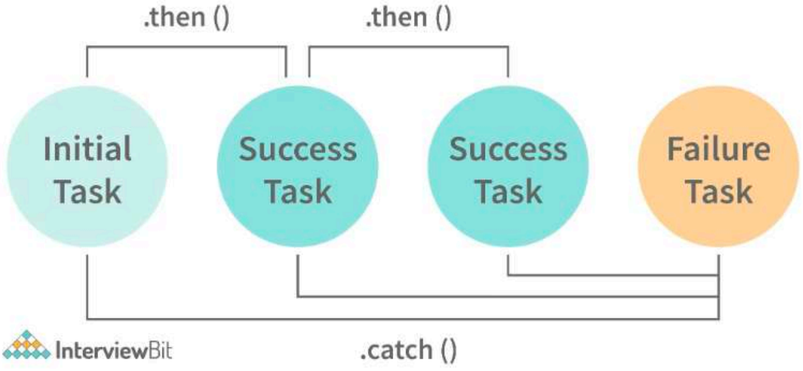
We can consume any promise by attaching then() and catch() methods to the consumer.



```
        i++;
      }
      resolve("Sum has been calculated: "+sum);
    }
  })
}
```

In the code above, we are calculating the sum of three elements, if the length of the elements array is more than 3, a promise is rejected, or else the promise is resolved and the sum is returned.

We can consume any promise by attaching then() and catch() methods to the consumer.



then() method is used to access the result when the promise is fulfilled.

catch() method is used to access the result/error when the promise is rejected. In the code below, we are consuming the promise:

```
sumOfThreeElements(4, 5, 6)
.then(result=> console.log(result))
.catch(error=> console.log(error));
// In the code above, the promise is fulfilled so the then() method gets executed

sumOfThreeElements(7, 0, 33, 41)
.then(result => console.log(result))
.catch(error=> console.log(error));
// In the code above, the promise is rejected hence the catch() method gets executed
```

38. What are classes in javascript?

Introduced in the ES6 version, classes are nothing but syntactic sugars for constructor functions. They provide a new way of declaring constructor functions in javascript. Below are the examples of how classes are declared and used:

```
// Before ES6 version, using constructor functions
function Student(name,rollNumber,grade,section){
  this.name = name;
  this.rollNumber = rollNumber;
  this.grade = grade;
  this.section = section;
}

// Way to add methods to a constructor function
Student.prototype.getDetails = function(){
  return 'Name: ${this.name}, Roll no: ${this.rollNumber}, Grade: ${this.grade}, Section:${this.section}';
}


let student1 = new Student("Vivek", 354, "6th", "A");
student1.getDetails();
// Returns Name: Vivek, Roll no:354, Grade: 6th, Section:A

// ES6 version classes
class Student{
  constructor(name,rollNumber,grade,section){
    this.name = name;
    this.rollNumber = rollNumber;
    this.grade = grade;
    this.section = section;
  }

  // Methods can be directly added inside the class
  getDetails(){
    return 'Name: ${this.name}, Roll no: ${this.rollNumber}, Grade:${this.grade}, Section:${this.section}';
  }
}

let student2 = new Student("Garry", 673, "7th", "C");
student2.getDetails();
// Returns Name: Garry, Roll no:673, Grade: 7th, Section:C
```

```
.catch(error=> console.log(error));
// In the code above, the promise is rejected hence the catch() method gets executed
```

38. What are classes in javascript?

Introduced in the ES6 version, classes are nothing but syntactic sugars for constructor functions. They provide a new way of declaring constructor functions in javascript. Below are the examples of how classes are declared and used:

```
// Before ES6 version, using constructor functions
function Student(name,rollNumber,grade,section){
  this.name = name;
  this.rollNumber = rollNumber;
  this.grade = grade;
  this.section = section;
}

// Way to add methods to a constructor function
Student.prototype.getDetails = function(){
  return 'Name: ${this.name}, Roll no: ${this.rollNumber}, Grade: ${this.grade}, Section:${this.section}';
}


let student1 = new Student("Vivek", 354, "6th", "A");
student1.getDetails();
// Returns Name: Vivek, Roll no:354, Grade: 6th, Section:A

// ES6 version classes
class Student{
  constructor(name,rollNumber,grade,section){
    this.name = name;
    this.rollNumber = rollNumber;
    this.grade = grade;
    this.section = section;
  }

  // Methods can be directly added inside the class
  getDetails(){
    return 'Name: ${this.name}, Roll no: ${this.rollNumber}, Grade:${this.grade}, Section:${this.section}';
  }
}

let student2 = new Student("Garry", 673, "7th", "C");
student2.getDetails();
// Returns Name: Garry, Roll no:673, Grade: 7th, Section:C
```

Key points to remember about classes:

- Unlike functions, classes are not hoisted. A class cannot be used before it is declared.
- A class can inherit properties and methods from other classes by using the extend keyword.
- All the syntaxes inside the class must follow the strict mode('use strict') of javascript. An error will be thrown if the strict mode rules are not followed.

39. What are generator functions?

Introduced in the ES6 version, generator functions are a special class of functions.

They can be stopped midway and then continue from where they had stopped.

Generator functions are declared with the **function*** keyword instead of the normal **function** keyword:

```
function* genFunc(){
  // Perform operation
}
```

In normal functions, we use the **return** keyword to return a value and as soon as the return statement gets executed, the function execution stops:

```
function normalFunc(){
  return 22;
  console.log(2);  // This line of code does not get executed
}
```

In the case of generator functions, when called, they do not execute the code, instead, they return a **generator object**. This generator object handles the execution.


```
function* genFunc(){
```

39. What are generator functions?

Introduced in the ES6 version, generator functions are a special class of functions.

They can be stopped midway and then continue from where they had stopped.

Generator functions are declared with the **function*** keyword instead of the normal **function** keyword:

```
function* genFunc(){
  // Perform operation
}
```

In normal functions, we use the **return** keyword to return a value and as soon as the return statement gets executed, the function execution stops:

```
function normalFunc(){
  return 22;
  console.log(2); // This line of code does not get executed
}
```

In the case of generator functions, when called, they do not execute the code, instead, they return a **generator object**. This generator object handles the execution.

```
function* genFunc(){
  yield 3;
  yield 4;
}
genFunc(); // Returns Object [Generator] {}
```

The generator object consists of a method called **next()**, this method when called, executes the code until the nearest **yield** statement, and returns the yield value.

For example, if we run the next() method on the above code:

```
genFunc().next(); // Returns {value: 3, done:false}
```

As one can see the next method returns an object consisting of a **value** and **done** properties. Value property represents the yielded value. Done property tells us whether the function code is finished or not. (Returns true if finished).

Generator functions are used to return iterators. Let's see an example where an iterator is returned:

```
function* iteratorFunc() {
  let count = 0;
  for (let i = 0; i < 2; i++) {
      count++;
      yield i;
  }
  return count;
}

let iterator = iteratorFunc();
console.log(iterator.next()); // {value:0,done:false}
console.log(iterator.next()); // {value:1,done:false}
console.log(iterator.next()); // {value:2,done:true}
```

As you can see in the code above, the last line returns **done:true**, since the code reaches the return statement.

40. Explain WeakSet in javascript.

In javascript, a Set is a collection of unique and ordered elements. Just like Set, WeakSet is also a collection of unique and ordered elements with some key differences:

- Weakset contains only objects and no other type.
- An object inside the weakset is referenced weakly. This means, that if the object inside the weakset does not have a reference, it will be garbage collected.
- Unlike Set, WeakSet only has three methods, **add()** , **delete()** and **has()** .

```
const newSet = new Set([4, 5, 6, 7]);
console.log(newSet);// Outputs Set {4,5,6,7}

const newSet2 = new WeakSet([3, 4, 5]); //Throws an error


let obj1 = {message:"Hello world"};
const newSet3 = new WeakSet([obj1]);
console.log(newSet3.has(obj1)); // true
```

41. Why do we use callbacks?

A callback function is a method that is sent as an input to another function (now let us name this other function "thisFunction"), and it is performed inside the thisFunction after the function has completed execution.

JavaScript is a scripting language that is based on events. Instead of waiting for a reply before continuing, JavaScript will continue to run while monitoring for additional events. Callbacks are a technique of [illegible] code has completed its execution.

```
let obj1 = {message:"Hello world"};
const newSet3 = new WeakSet([obj1]);
console.log(newSet3.has(obj1)); // true
```

41. Why do we use callbacks?

A callback function is a method that is sent as an input to another function (now let us name this other function "thisFunction"), and it is performed inside the thisFunction after the function has completed execution.

JavaScript is a scripting language that is based on events. Instead of waiting for a reply before continuing, JavaScript will continue to run while monitoring for additional events. Callbacks are a technique of ensuring that a particular code does not run until another code has completed its execution.

42. Explain WeakMap in javascript.

In javascript, Map is used to store key-value pairs. The key-value pairs can be of both primitive and non-primitive types. WeakMap is similar to Map with key differences:

- The keys and values in weakmap should always be an object.
- If there are no references to the object, the object will be garbage collected.

```
const map1 = new Map();
map1.set('Value', 1);

const map2 = new WeakMap();
map2.set('Value', 2.3); // Throws an error

let obj = {name:"Vivek"};
const map3 = new WeakMap();
map3.set(obj, {age:23});
```

43. What is Object Destructuring?

Object destructuring is a new way to extract elements from an object or an array.

- **Object destructuring:** Before ES6 version:

```
const classDetails = {
  strength: 78,
  benches: 39,
  blackBoard:1
}

const classStrength = classDetails.strength;
const classBenches = classDetails.benches;
const classBlackBoard = classDetails.blackBoard;
```

The same example using object destructuring:

```
const classDetails = {
  strength: 78,
  benches: 39,
  blackBoard:1
}

const {strength:classStrength, benches:classBenches,blackBoard:classBlackBoard} = classDetails;

console.log(classStrength); // Outputs 78
console.log(classBenches); // Outputs 39
console.log(classBlackBoard); // Outputs 1
```

As one can see, using object destructuring we have extracted all the elements inside an object in one line of code. If we want our new variable to have the same name as the property of an object we can remove the colon:

```
const {strength:strength} = classDetails;
// The above line of code can be written as:
const {strength} = classDetails;
```

- **Array destructuring:** Before ES6 version:

```
const arr = [1, 2, 3, 4];
const first = arr[0];
const second = arr[1];
const third = arr[2];
const fourth = arr[3];
```

The same example using object destructuring:

```
const arr = [1, 2, 3, 4];
const [first,second,third,fourth] = arr;
console.log(first); // Outputs 1
console.log(second); // Outputs 2
console.log(third); // Outputs 3
console.log(fourth); // Outputs 4
```

44. Difference between prototypal and classical inheritance

Programers build objects, which are representations of real-time entities, i
two sorts of abstractions. A class is a generalization of an object, whereas

```
console.log(third); // Outputs 3
console.log(fourth); // Outputs 4
```

44. Difference between prototypal and classical inheritance

Programers build objects, which are representations of real-time entities, in traditional OO programming. Classes and objects are the two sorts of abstractions. A class is a generalization of an object, whereas an object is an abstraction of an actual thing. A Vehicle, for example, is a specialization of a Car. As a result, automobiles (class) are descended from vehicles (object).

Classical inheritance differs from prototypal inheritance in that classical inheritance is confined to classes that inherit from those remaining classes, but prototypal inheritance allows any object to be cloned via an object linking method. Despite going into too many specifics, a prototype essentially serves as a template for those other objects, whether they extend the parent object or not.

45. What is a Temporal Dead Zone?

Temporal Dead Zone is a behaviour that occurs with variables declared using **let** and **const** keywords. It is a behaviour where we try to access a variable before it is initialized. Examples of temporal dead zone:

```
x = 23; // Gives reference error

let x;

function anotherRandomFunc(){
  message = "Hello"; // Throws a reference error

  let message;
}
anotherRandomFunc();
```

In the code above, both in the global scope and functional scope, we are trying to access variables that have not been declared yet. This is called the **Temporal Dead Zone**.

46. What do you mean by JavaScript Design Patterns?

JavaScript design patterns are repeatable approaches for errors that arise sometimes when building JavaScript browser applications. They truly assist us in making our code more stable.

They are divided mainly into 3 categories

1. Creational Design Pattern
2. Structural Design Pattern
3. Behavioral Design Pattern.

- **Creational Design Pattern:** The object generation mechanism is addressed by the JavaScript Creational Design Pattern. They aim to make items that are appropriate for a certain scenario.
- **Structural Design Pattern:** The JavaScript Structural Design Pattern explains how the classes and objects we've generated so far can be combined to construct bigger frameworks. This pattern makes it easier to create relationships between items by defining a straightforward way to do so.
- **Behavioral Design Pattern:** This design pattern highlights typical patterns of communication between objects in JavaScript. As a result, the communication may be carried out with greater freedom.

47. Is JavaScript a pass-by-reference or pass-by-value language?

The variable's data is always a reference for objects, hence it's always pass by value. As a result, if you supply an object and alter its members inside the method, the changes continue outside of it. It appears to be pass by reference in this case. However, if you modify the values of the object variable, the change will not last, demonstrating that it is indeed passed by value.

48. Difference between Async/Await and Generators usage to achieve the same functionality.

- Generator functions are run by their generator yield by yield which means one output at a time, whereas Async-await functions are executed sequentially one after another.
- Async/await provides a certain use case for Generators easier to execute.
- The output result of the Generator function is always value: X, done: Boolean, but the return value of the Async function is always an assurance or throws an error.

49. What are the primitive data types in JavaScript?

A primitive is a data type that isn't composed of other data types. It's only capable of displaying one value at a time. By definition, every primitive is a built-in data type (the compiler must be knowledgeable of them) nevertheless, not all built-in datasets are primitives. In JavaScript, there are 5 different forms of basic data. The following values are available:

1. Boolean
2. Undefined
3. Null
4. Number
5. String

50. What is the role of deferred scripts in JavaScript?

The processing of HTML code while the page loads are disabled by nature till the script hasn't halted. Your page will be affected if your network is a bit slow, or if the script is very hefty. When you use Deferred, the script waits for the HTML parser to finish before executing it. This reduces the time it takes for web pages to load, allowing them to appear more quickly.

51. What has to be done in order to put Lexical Scoping into

To support lexical scoping, a JavaScript function object's internal state mu

2. Undefined
3. Null
4. Number
5. String

50. What is the role of deferred scripts in JavaScript?

The processing of HTML code while the page loads are disabled by nature till the script hasn't halted. Your page will be affected if your network is a bit slow, or if the script is very hefty. When you use Deferred, the script waits for the HTML parser to finish before executing it. This reduces the time it takes for web pages to load, allowing them to appear more quickly.

51. What has to be done in order to put Lexical Scoping into practice?

To support lexical scoping, a JavaScript function object's internal state must include not just the function's code but also a reference to the current scope chain.

52. What is the purpose of the following JavaScript code?

```
var scope = "global scope";
function check()
{
    var scope = "local scope";
    function f()
  {
        return scope;
    }
    return f;
}
```

Every executing function, code block, and script as a whole in JavaScript has a related object known as the Lexical Environment. The preceding code line returns the value in scope.

JavaScript Coding Interview Questions

53. What is the output of the following code?

```
const b = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

for (let i = 0; i < 10; i++) {
  setTimeout(() => console.log(b[i]), 1000);
}

for (var i = 0; i < 10; i++) {
  setTimeout(() => console.log(b[i]), 1000);
}
```

Ans.

```
1
2
3
4
5
6
7
8
9
10
undefined
undefined
undefined
undefined
undefined
undefined
undefined
undefined
undefined
undefined
```

Conclusion

It is preferable to keep the JavaScript, CSS, and HTML in distinct Separate 'javascript' files. Dividing the code and HTML sections will make them easier to understand and deal with. This strategy is also simpler for several programmers to use at the same time. JavaScript code is simple to update. Numerous pages can utilize the same group of JavaScript Codes. If we utilize External JavaScript scripts and need to alter the code, we must do it just once. So that we may utilize a number and maintain it much more easily. Remember that professional experience and expertise are only one aspect of recruitment. Previous experience and personal skills are both vital in landing (or finding the ideal applicant for the job.

Remember that many JavaScript structured interviews are free and have no one proper answer. Interviewers would like to know why you answered the way you did, not if you remembered the answer. Explain your answer process and be prepared to address it. If you're looking to further enhance your JavaScript skills, consider enrolling in this free JavaScript course by Scaler Topics to gain hands-on experience and improve your problem-solving abilities.

Recommended Resources

- JavaScript Cheat Sheet: Basics to Advanced(2022)

- Mock Interview
- InterviewBit - No.1 Resource for Tech Interview Preparation

54. In JavaScript, how do you turn an Object into an Array []?

```
let obj = { id: "1", name: "user22", age: "26", work: "programmer" };

//Method 1: Convert the keys to Array using - Object.keys()
console.log(Object.keys(obj));
// ["id", "name", "age", "work"]

// Method 2 Converts the Values to Array using - Object.values()
console.log(Object.values(obj));
// ["1", "user22r", "26", "programmer"]

// Method 3 Converts both keys and values using - Object.entries()
console.log(Object.entries(obj));
//[["id", "1"],["name", "user22"],["age", "26"],["work", "programmer"]]
```

55. Write the code to find the vowels

```
const findVowels = str => {
  let count = 0
  const vowels = ['a', 'e', 'i', 'o', 'u']
  for(let char of str.toLowerCase()) {
    if(vowels.includes(char)) {
      count++
    }
  }
  return count
}
```

56. Write the code given If two strings are anagrams of one another, then return true.

```
var firstWord = "Deepak";
var secondWord = "Aman";

isAnagram(wordOne, wordTwo); // true

function isAnagram(one, two) {
  //Change both words to lowercase for case insensitivity..
  var a = one.toLowerCase();
  var b = two.toLowerCase();

  // Sort the strings, then combine the array to a string. Examine the outcomes.
  a = a.split("").sort().join("");
  b = b.split("").sort().join("");

  return a === b;
}
```

57. Write the code for dynamically inserting new components.

```
<html>
<head>
<title>inserting new components dynamically</title>
<script type="text/javascript">
    function addNode () { var newP = document. createElement("p");
    var textNode = document.createTextNode(" This is other node");
    newP.appendChild(textNode); document.getElementById("parent1").appendChild(newP); }
</script>
</head>
<body> <p id="parent1">firstP<p> </body>
</html>
```

58. Implement a function that returns an updated array with r right rotations on an array of integers a .

Example:

Given the following array: **[2,3,4,5,7]**
Perform **3** right rotations:
First rotation : [7,2,3,4,5] , Second rotation : [5,7,2,3,4] and, Third rotation: [4,5,7,2,3]

return **[4,5,7,2,3]**

Answer:

```
function rotateRight(arr,rotations){
  if(rotations == 0) return arr;
  for(let i = 0; i < rotations;i++){
    let element = arr.pop();
    arr.unshift(element);
  }
  return arr;
}
rotateRight([2, 3, 4, 5, 7], 3); // Return [4,5,7,2,3]
```

```
</head>
<body> <p id="parent1">firstP<p> </body>
</html>
```

58. Implement a function that returns an updated array with r right rotations on an array of integers a .

Example:

Given the following array: **[2,3,4,5,7]**
Perform **3** right rotations:
First rotation : [7,2,3,4,5] , Second rotation : [5,7,2,3,4] and, Third rotation: [4,5,7,2,3]

return **[4,5,7,2,3]**

Answer:

```
function rotateRight(arr,rotations){
  if(rotations == 0) return arr;
  for(let i = 0; i < rotations;i++){
    let element = arr.pop();
    arr.unshift(element);
  }
  return arr;
}
rotateRight([2, 3, 4, 5, 7], 3); // Return [4,5,7,2,3]
rotateRight([44, 1, 22, 111], 5); // Returns [111,44,1,22]
```

59. Write a function that performs binary search on a sorted array.

```
function binarySearch(arr,value,startPos,endPos){
  if(startPos > endPos) return -1;

  let middleIndex = Math.floor(startPos+endPos)/2;

  if(arr[middleIndex] === value) return middleIndex;

  elsif(arr[middleIndex] > value){
    return binarySearch(arr,value,startPos,middleIndex-1);
  }
  else{
    return binarySearch(arr,value,middleIndex+1,endPos);
  }
}
```

60. Guess the outputs of the following code:

****Note - Code 2 and Code 3 require you to modify the code, instead of guessing the output.**

```
// Code 1

(function(a){
  return (function(){
    console.log(a);
    a = 23;
  })()
})(45);

// Code 2

// Each time bigFunc is called, an array of size 700 is being created,
// Modify the code so that we don't create the same array again and again

function bigFunc(element){
  let newArray = new Array(700).fill('♥');
  return newArray[element];
}

console.log(bigFunc(599)); // Array is created
console.log(bigFunc(670)); // Array is created again

// Code 3

// The following code outputs 2 and 2 after waiting for one second
// Modify the code to output 0 and 1 after one second.

function randomFunc(){
  for(var i = 0; i < 2; i++){
    setTimeout(()=> console.log(i),1000);
  }
}
randomFunc();
```

Answers -

Code 1 - Outputs **45**.

```
}
```

60. Guess the outputs of the following code:

****Note - Code 2 and Code 3 require you to modify the code, instead of guessing the output.**

```
// Code 1

(function(a){
  return (function(){
    console.log(a);
    a = 23;
  })()
})(45);

// Code 2

// Each time bigFunc is called, an array of size 700 is being created,
// Modify the code so that we don't create the same array again and again

function bigFunc(element){
  let newArray = new Array(700).fill('♥');
  return newArray[element];
}

console.log(bigFunc(599)); // Array is created
console.log(bigFunc(670)); // Array is created again

// Code 3

// The following code outputs 2 and 2 after waiting for one second
// Modify the code to output 0 and 1 after one second.

function randomFunc(){
  for(var i = 0; i < 2; i++){
    setTimeout(()=> console.log(i),1000);
  }
}
randomFunc();
```

Answers -

Code 1 - Outputs **45**.

Even though a is defined in the outer function, due to closure the inner functions have access to it.

Code 2 - This code can be modified by using closures,

```
function bigFunc(){
  let newArray = new Array(700).fill('♥');
  return (element) => newArray[element];
}

let getElement = bigFunc(); // Array is created only once
getElement(599);
getElement(670);
```

Code 3 - Can be modified in two ways:

Using **let** keyword:

```
function randomFunc(){
  for(let i = 0; i < 2; i++){
    setTimeout(()=> console.log(i),1000);
  }
}
randomFunc();
```

Using closure:

```
function randomFunc(){
  for(var i = 0; i < 2; i++){
  (function(i){
      setTimeout(()=>console.log(i),1000);
    })(i);
  }
}
randomFunc();
```

61. Guess the outputs of the following code:

```
// Code 1

  let hero = {
    powerLevel: 99,
    getPower(){
      return this.powerLevel;
```

```
  }
}
randomFunc();
```

61. Guess the outputs of the following code:

```
// Code 1

  let hero = {
    powerLevel: 99,
    getPower(){
      return this.powerLevel;
    }
  }

  let getPower = hero.getPower;

  let hero2 = {powerLevel:42};
  console.log(getPower());
  console.log(getPower.apply(hero2));



  // Code 2

  const a = function(){
    console.log(this);

    const b = {
      func1: function(){
        console.log(this);
      }
    }

    const c = {
      func2: ()=>{
        console.log(this);
      }
    }

    b.func1();
    c.func2();
  }

  a();



  // Code 3

  const b = {
    name:"Vivek",
    f: function(){
      var self = this;
      console.log(this.name);
      (function(){
        console.log(this.name);
        console.log(self.name);
      })();
    }
  }
  b.f();
```

Answers:

Code 1 - Output in the following order:

```
undefined
42
```

Reason - The first output is **undefined** since when the function is invoked, it is invoked referencing the global object:

```
window.getPower() = getPower();
```

Code 2 - Outputs in the following order:

```
global/window object
object "b"
global/window object
```

Since we are using the arrow function inside **func2, this** keyword refers to the global object.

Code 3 - Outputs in the following order:

```
"Vivek"
```

```
undefined
42
```

Reason - The first output is **undefined** since when the function is invoked, it is invoked referencing the global object:

```
window.getPower() = getPower();
```

Code 2 - Outputs in the following order:

```
global/window object
object "b"
global/window object
```

Since we are using the arrow function inside **func2, this** keyword refers to the global object.

Code 3 - Outputs in the following order:

```
"Vivek"
undefined
"Vivek"
```

Only in the IIFE inside the function **f**, **this** keyword refers to the global/window object.

62. Guess the output of the following code:

```
var x = 23;

(function(){
  var x = 43;
  (function random(){
    x++;
    console.log(x);
    var x = 21;
  })();
})();
```

Answer:

Output is **NaN**.

random() function has functional scope since x is declared and hoisted in the functional scope.

Rewriting the random function will give a better idea about the output:

```
function random(){
  var x; // x is hoisted
  x++; // x is not a number since it is not initialized yet
  console.log(x); // Outputs NaN
  x = 21; // Initialization of x
}
```

63. Guess the outputs of the following code:

```
// Code 1:

let x= {}, y = {name:"Ronny"},z = {name:"John"};
x[y] = {name:"Vivek"};
x[z] = {name:"Akki"};
console.log(x[y]);

// Code 2:

function runFunc(){
  console.log("1" + 1);
  console.log("A" - 1);
  console.log(2 + "-2" + "2");
  console.log("Hello" - "World" + 78);
  console.log("Hello"+ "78");
}
runFunc();

// Code 3:

let a = 0;
let b = false;
console.log((a == b));
console.log((a === b));
```

Answers:

Code 1 - Output will be **{name: "Akki"}.**

Adding objects as properties of another object should be done carefully.

Writing x[y] = {name:"Vivek"}, is same as writing x['object Object'] = {name:

```
// Code 3:

let a = 0;
let b = false;
console.log((a == b));
console.log((a === b));
```

Answers:

Code 1 - Output will be **{name: "Akki"}.**

Adding objects as properties of another object should be done carefully.

Writing **x[y] = {name:"Vivek"}** , is same as writing **x['object Object'] = {name:"Vivek"}** ,

While setting a property of an object, **javascript coerces the parameter into a string.**

Therefore, since **y** is an object, it will be converted to **'object Object'.**

Both x[y] and x[z] are referencing the same property.

Code 2 - Outputs in the following order:

```
11
Nan
2-22
NaN
Hello78
```

Code 3 - Output in the following order due to equality coercion:

```
true
false
```

64. Guess the outputs of the following codes:

```
// Code 1:

function func1(){
  setTimeout(()=>{
    console.log(x);
    console.log(y);
  },3000);

  var x = 2;
  let y = 12;
}
func1();

// Code 2:

function func2(){
  for(var i = 0; i < 3; i++){
    setTimeout(()=> console.log(i),2000);
}
}
func2();

// Code 3:

(function(){
  setTimeout(()=> console.log(1),2000);
  console.log(2);
  setTimeout(()=> console.log(3),0);
  console.log(4);
})();
```

Answers:

- **Code 1** - Outputs **2** and **12**. Since, even though **let** variables are not hoisted, due to the async nature of javascript, the complete function code runs before the setTimeout function. Therefore, it has access to both x and y.
- **Code 2** - Outputs **3**, three times since variable declared with **var** keyword does not have block scope. Also, inside the for loop, the variable i is incremented first and then checked.
- **Code 3** - Output in the following order:

```
2
4
3
1  // After two seconds
```

Even though the second timeout function has a waiting time of zero seconds, the javascript engine always evaluates the setTimeout function using the Web API, and therefore, the complete function executes before the setTimeout function can execute.

JavaScript MCQ

1.Argument class is

Let's discuss some common questions that you should prepare for the interviews. These questions will help clear the interviews, especially for the frontend development role.

1. Are JavaScript and Java related?

Java is an object-oriented programming language, while JavaScript is a client-side scripting language. Both of them are different from each other. Here are main difference point between them:

Java	JavaScript
Statically typed, compiled and then interpreted	Dynamically typed, interpreted
Runs on JVM (Java Virtual Machine)	Runs in a browser or Node.js
Class-based inheritance	Prototype-based inheritance
Compiled into bytecode and then bytcdode is run by the interpreter	Interpreted directly by browser
Multi-threading and concurrency supported	Single-threaded, async with event loop
Strongly typed	Weakly typed
Fully object-oriented	Supports object-oriented but more flexible
Automatic garbage collection (JVM)	Automatic garbage collection (browser)

2. What are Data Types in JavaScript?

JavaScript data types are categorized into two parts i.e. primitive and non-primitive types.

- **Primitive Data Type:** The predefined data types provided by JavaScript language are known as primitive data type. Primitive data types are also known as in-built data types.
  - Numbers
  - Strings
  - Boolean
  - Symbol
  - Undefined
  - Null
  - BigInt
- **Non-Premitive Data Type:** The data types that are derived from primitive data types are known as non-primitive data types. It is also known as derived data types or reference data types.
  - Objects
  - Functions
  - Arrays

3. What is the latest version of JavaScript?

The newest version of JavaScript is called **ECMAScript 2023 (ES2023)**. It came out in June 2023.

4. What would be the result of 3+2+"7"?

Here, 3 and 2 behave like an integer, and "7" behaves like a string. So 3 plus 2 will be 5. Then the output will be 5+"7" = 57.

5. What is the use of the isNaN function?

The number isNan function determines whether the passed value is NaN (Not a number) and is of the type "Number". In JavaScript, the value NaN is considered a type of number. It returns true if the argument is not a number, else it returns false.

6. Which is faster in JavaScript and ASP script?

JavaScript is faster compared to ASP Script. JavaScript is a client-side scripting language and does

The number isNan function determines whether the passed value is NaN (Not a number) and is of the type "Number". In JavaScript, the value NaN is considered a type of number. It returns true if the argument is not a number, else it returns false.

6. Which is faster in JavaScript and ASP script?

JavaScript is faster compared to ASP Script. JavaScript is a client-side scripting language and does not depend on the server to execute. The ASP script is a server-side scripting language always dependable on the server.

7. What is negative infinity?

The negative infinity is a constant value represents the lowest available value. It means that no other number is lesser than this value. It can be generate using a self-made function or by an arithmetic operation. JavaScript shows the NEGATIVE_INFINITY value as -Infinity.

8. Is it possible to break JavaScript Code into several lines?

Yes, it is possible to break the JavaScript code into several lines in a string statement. It can be broken by using the **'\n'** (backslash n).

Example:

```
console.log("A Online Computer Science Portal\n for Geeks")
```

The code-breaking line is avoid by JavaScript which is not preferable.

```
let gfg= 10, GFG = 5,
Geeks =
gfg + GFG;
```

9. What are "truthy" and "falsy" values in JavaScript

- **Falsy**: false, 0, "" (empty string), null, undefined, NaN.
- **Truthy**: Everything else (e.g., any non-empty string, any non-zero number, objects, arrays).

10. What are undeclared and undefined variables?

- Undefined**:** It occurs when a variable is declare but not assign any value. Undefined is not a keyword.
- Undeclared**:** It occurs when we try to access any variable which is not initialize or declare earlier using the var or const keyword. If we use 'typeof' operator to get the value of an undeclare variable, we will face the runtime error with the return value as "undefined". The scope of the undeclare variables is always global.

11. Write a JavaScript code for adding new elements dynamically.

```
<html>
<head>
</head>
<body>
    <button onclick="create()">
        Click Here!
    </button>

    <script>
        function create() {
            let geeks = document.createElement('geeks');
            geeks.textContent = "Geeksforgeeks";
            geeks.setAttribute('class', 'note');
            document.body.appendChild(geeks);
        }
    </script>
</body>
</html>
```

12. What are global variables? How are these variables declared, and what are the problems associated with them?

```
    </script>
</body>
</html>
```

12. What are global variables? How are these variables declared, and what are the problems associated with them?

In contrast, global variables are the variables that define outside of functions. These variables have a global scope, so they can be used by any function without passing them to the function as parameters.

Example:

```
let petName = "Rocky"; // Global Variable
myFunction();

function myFunction() {
    console.log("Inside myFunction - Type of petName:", typeof petName);
    console.log("Inside myFunction - petName:", petName);
}

console.log("Outside myFunction - Type of petName:", typeof petName);
console.log("Outside myFunction - petName:", petName);
```

Output

```
Inside myFunction - Type of petName: string
Inside myFunction - petName: Rocky
Outside myFunction - Type of petName: string
Outside myFunction - petName: Rocky
```

It is difficult to debug and test the code that relies on global variables.

13. What do you mean by NULL in JavaScript?

The NULL value represents that no value or no object. It is known as empty value/object.

14. How to delete property-specific values?

The **delete keyword** deletes the whole property and all the values at once like

```
let gfg={Course: "DSA", Duration:30};
delete gfg.Course;
```

15. What is the difference between null and undefined in JavaScript?

A deeper comparison between the two special values, null (intentional absence of value) and undefined (uninitialized variables).

16. What is a prompt box?

The prompt box is a dialog box with an optional message prompting the user to input some text. It is often used if the user wants to input a value before entering a page. It returns a string containing the text entered by the user, or null.

17. What is the 'this' keyword in JavaScript?

Functions in JavaScript are essential objects. Like objects, it can be assign to variables, pass to other functions, and return from functions. And much like objects, they have their own properties. 'this' stores the current execution context of the JavaScript program. Thus, when it use inside a function, the value of 'this' will change depending on how the function is defined, how it is invoked, and the default execution context.

18. Explain the working of timers in JavaScript. Also explain the drawbacks of using the timer, if any.

The timer executes some specific code at a specific time or any small amount of code in repetition to do that you need to use the functions **setTimout, setInterval,** and **clearInterval**. If the JavaScript code sets the timer to 2 minutes and when the times are up then the page displays an alert message "times up". The **setTimeout()** method calls a function or evaluates an expression after a specified number of milliseconds.

19. What is the difference between ViewState and SessionState?

the value of 'this' will change depending on how the function is defined, how it is invoked, and the default execution context.

18. Explain the working of timers in JavaScript. Also explain the drawbacks of using the timer, if any.

The timer executes some specific code at a specific time or any small amount of code in repetition to do that you need to use the functions setTimout, setInterval, and clearInterval. If the JavaScript code sets the timer to 2 minutes and when the times are up then the page displays an alert message "times up". The **setTimeout()** method calls a function or evaluates an expression after a specified number of milliseconds.

19. What is the difference between ViewState and SessionState?

- **ViewState:** It is specific to a single page in a session.
- **SessionState:** It is user specific that can access all the data on the web pages.

20. How to submit a form using JavaScript?

You can use document.form[0].submit()method to submit the form in JavaScript.

21. Does JavaScript support automatic type conversion?

Yes, JavaScript supports automatic type conversion.

22. What is a template literal in JavaScript?

A template literal in JavaScript is a way to define strings that allow embedded expressions and multi-line formatting. It uses backticks (`) instead of quotes and supports `${}` for embedding variables or expressions inside the string.

23. What is a higher-order function in JavaScript?

A higher-order function in JavaScript is a function that either takes one or more functions as arguments, or returns a function as its result. These functions allow for more abstract and reusable code, enabling functional programming patterns

For example, map() and filter() are higher-order functions because they take callback functions as arguments.

JavaScript Intermediate Interview Questions

24. What are all the looping structures in JavaScript?

- **while loop:** A while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement.
- **for loop:** A for loop provides a concise way of writing the loop structure. Unlike a while loop, for statement consumes the initialization, condition and increment/decrement in one line thereby providing a shorter, easy to debug structure of looping.
- **do while:** A do-while loop is similar to while loop with the only difference that it checks the condition after executing the statements, and therefore is an example of Exit Control Loop.

25. What is lexical scope in JavaScript?

Lexical scope in JavaScript refers to the way variables are resolved based on their location in the source code. A variable's scope is determined by the position of the code where it is defined, and it is accessible to any nested functions or blocks. This means that functions have access to variables in their own scope and the outer (lexical) scopes, but not to variables in inner scopes.

```
let outer = "I am outside!";
function inner() {
    console.log(outer);
}
inner();
```

In this example, inner() can access the outer variable because of lexical scoping.

26. How does lexical scoping work with the `this` keyword in JavaScript?

In JavaScript, lexical scoping primarily applies to variable resolution, while the behavior of the this keyword is determined by how a function is called, not by its position in the code. The value of this is dynamically determined at runtime based on the function's context (e.g., whether it's called as a method, in a global context, or with call, apply, or bind).

```
const obj = {
    name: "JavaScript",
```

```
inner();
```

In this example, inner() can access the outer variable because of lexical scoping.

26. How does lexical scoping work with the `this` keyword in JavaScript?

In JavaScript, lexical scoping primarily applies to variable resolution, while the behavior of the this keyword is determined by how a function is called, not by its position in the code. The value of this is dynamically determined at runtime based on the function's context (e.g., whether it's called as a method, in a global context, or with call, apply, or bind).

```
const obj = {
    name: "JavaScript",
    greet: function () {
        console.log(this.name);
    }
};
obj.greet(); // "JavaScript"
```

Here, this refers to obj because the function is called as a method of the object. Lexical scoping affects variable lookups but doesn't alter how this behaves.

27. Explain how to read and write a file using JavaScript?

The readFile() functions is used for reading operation.

```
readFile( Path, Options, Callback)
```

The writeFile() functions is used for writing operation.

```
writeFile( Path, Data, Callback)
```

28. What is called Variable typing in JavaScript?

The **variable typing** is the type of variable used to store a number and using that same variable to assign a "string".

```
Geeks = 42;
Geeks = "GeeksforGeeks";
```

29. What is hoisting in JavaScript?

Hoisting in JavaScript is the behavior where variable and function declarations are moved to the top of their containing scope during compilation, before the code is executed. This means you can reference variables and functions before they are declared in the code. However, only declarations are hoisted, not initializations.

```
console.log(a); // undefined
var a = 5;
```

In this case, the declaration of a is hoisted, but its value (5) is not assigned until the code execution reaches that line. Hoisting applies differently for var, let, const, and function declarations.

30. How to convert the string of any base to integer in JavaScript?

In JavaScript, parseInt() function is used to convert the string to an integer. This function returns an integer of base which is specified in second argument of parseInt() function. The parseInt() function returns Nan (not a number) when the string doesn't contain number.

31. Explain how to detect the operating system on the client machine?

To detect the operating system on the client machine, one can simply use navigator.appVersion or navigator.userAgent property. The Navigator appVersion property is a read-only property and it returns the string that represents the version information of the browser.

32. What are the types of Pop up boxes available in JavaScript?

There are three types of pop boxes available in JavaScript.

- Alert
- Confirm
- Prompt

33. What is the use of void(0) ?

The void(0) is used to call another method without refreshing the page during the calling time parameter "zero" will be passed.

- Alert
- Confirm
- Prompt

33. What is the use of void(0) ?

The void(0) is used to call another method without refreshing the page during the calling time parameter “zero” will be passed.

34. What are JavaScript modules, and how do you import/export them?

JavaScript modules allow you to split your code into smaller, reusable pieces. They enable the export of variables, functions, or objects from one file and the import of them into another. To export an element, you use export (either named or default). To import it, you use import.

```
// In file1.js
export const greet = () => "Hello";

// In file2.js
import { greet } from './file1';
console.log(greet());
```

Modules help organize code and avoid global namespace pollution. They are natively supported in modern JavaScript through import and export statements.

35. What are WeakMap and WeakSet, and how are they different from Map and Set?

A **WeakMap** is a collection of key-value pairs where keys are objects and the values can be any data type. The key-value pairs in a WeakMap are "weakly" held, meaning if no other references to a key exist, the entry can be garbage collected. A **WeakSet** is a collection of unique objects, and like WeakMap, the objects are weakly held.

The main difference from **Map** and **Set** is that in **Map** and **Set**, entries are strongly held, meaning they prevent garbage collection, while in **WeakMap** and **WeakSet**, entries can be garbage collected if no other references to the objects exist.

36. What is the role of the setImmediate function in Node.js, and how is it different from setTimeout?

In Node.js, the `setImmediate()` function schedules a callback to be executed in the next iteration of the event loop, specifically after the current event loop phase (which includes I/O events). It's commonly used for deferring tasks to be executed once the current operation completes.

The key difference between `setImmediate()` and `setTimeout()` is that `setImmediate()` runs after the current event loop iteration, following I/O events but before any timers (such as `setTimeout()`). On the other hand, `setTimeout()` schedules tasks to run after a specified delay, which might not be precise if the event loop is busy, meaning it can be delayed by I/O or other tasks. While `setTimeout()` schedules tasks with a minimum delay, `setImmediate()` executes as soon as the event loop reaches a free point in the "check" phase, after I/O events have been processed.

JavaScript Interview Questions for Experienced

37. What is the ‘Strict’ mode in JavaScript and how can it be enabled?

Strict Mode is a new feature in ECMAScript 5 that allows you to place a program or a function in a “strict” operating context. This strict context prevents certain actions from being taken and throws more exceptions. The statement “use strict” instructs the browser to use the Strict mode, which is a reduced and safer feature set of JavaScript.

38. What are the advantages and disadvantages of using `async/await` over traditional callbacks or promises?

Advantages of async/await:

- **Improved Readability** : Async/await makes asynchronous code look like synchronous code, making it easier to read and maintain.
- **Simplified Error Handling** : With try/catch, error handling is more straightforward compared to .catch() with promises or callback-based error handling.
- **Avoids callback hell** : It eliminates deeply nested callbacks, reducing complexity in asynchronous logic.

Disadvantages of async/await:

- **Requires modern JavaScript** : It’s supported in ES2017 and above, so older environments may need transpiling.
- **Limited concurrency control** : Unlike promises with .all() or .race(), async/await can be less flexible for handling multiple parallel tasks.

more exceptions. The statement "use strict" instructs the browser to use the Strict mode, which is a reduced and safer feature set of JavaScript.

38. What are the advantages and disadvantages of using `async/await` over traditional callbacks or promises?

Advantages of async/await:

- **Improved Readability** : Async/await makes asynchronous code look like synchronous code, making it easier to read and maintain.
- **Simplified Error Handling** : With try/catch, error handling is more straightforward compared to .catch() with promises or callback-based error handling.
- **Avoids callback hell** : It eliminates deeply nested callbacks, reducing complexity in asynchronous logic.

Disadvantages of async/await:

- **Requires modern JavaScript** : It's supported in ES2017 and above, so older environments may need transpiling.
- **Limited concurrency control** : Unlike promises with .all() or .race(), async/await can be less flexible for handling multiple parallel tasks.

39. How to explain closures in JavaScript and when to use it?

The closure is created when a child functions to keep the environment of the parent's scope even after the parent's function has already executed. The Closure is a locally declared variable related to a function. The closure will provide better control over the code when using them.

```
function foo() {
    let b = 1;
    function inner() {
        return b;
    }
    return inner;
}
let get_func_inner = foo();

console.log(get_func_inner());
console.log(get_func_inner());
console.log(get_func_inner());
```

Output

```
1
1
1
```

40. What is the difference between call() and apply() methods ?

Both methods are used in a different situation

- **call() Method:** It calls the method, taking the owner object as argument. The keyword this refers to the 'owner' of the function or the object it belongs to. We can call a method that can be used on different objects.
- **apply() Method:** The apply() method is used to write methods, which can be used on different objects. It is different from the function call() because it takes arguments as an array.

41. How to target a particular frame from a hyperlink in JavaScript ?

This can be done by using the **target** attribute in the hyperlink. Like

```
<a href="/geeksforgeeks.htm" target="newframe">New Page</a>
```

42. Write the errors shown in JavaScript?

There are three different types of errors in JavaScript.

- Syntax error : A syntax error is an error in the syntax of a sequence of characters or tokens that are intended to be written in a particular programming language.
- **Logical error:** It is the most difficult error to be traced as it is the error on the logical part of the coding or logical error is a bug in a program that causes to operate incorrectly and terminate abnormally.
- Runtime Error : A runtime error is an error that occurs during the running of the program, also known as an exception.

43. What is the difference between JavaScript and Jscript?

to the 'owner' of the function or the object it belongs to. We can call a method that can be used on different objects.

- **apply() Method:** The apply() method is used to write methods, which can be used on different objects. It is different from the function call() because it takes arguments as an array.

41. How to target a particular frame from a hyperlink in JavaScript ?

This can be done by using the **target** attribute in the hyperlink. Like

```
<a href="/geeksforgeeks.htm" target="newframe">New Page</a>
```

42. Write the errors shown in JavaScript?

There are three different types of errors in JavaScript.

- Syntax error : A syntax error is an error in the syntax of a sequence of characters or tokens that are intended to be written in a particular programming language.
- **Logical error:** It is the most difficult error to be traced as it is the error on the logical part of the coding or logical error is a bug in a program that causes to operate incorrectly and terminate abnormally.
- Runtime Error : A runtime error is an error that occurs during the running of the program, also known as an exception.

43. What is the difference between JavaScript and Jscript?

JavaScript

- It is a scripting language developed by Netscape.
- It is used to design client and server-side applications.
- It is completely independent of Java language.

JScript

- It is a scripting language developed by Microsoft.
- It is used to design active online content for the word wide Web.

44. How many ways an HTML element can be accessed in JavaScript code?

There are four possible ways to access HTML elements in JavaScript which are:

- getElementById() Method: It is used to get the element by its id name.
- getElementsByClass() Method: It is used to get all the elements that have the given classname.
- getElementsByTagName() Method: It is used to get all the elements that have the given tag name.
- querySelector() Method: This function takes CSS style selector and returns the first selected element.

45. What is an event bubbling in JavaScript?

Consider a situation an element is present inside another element and both of them handle an event. When an event occurs in bubbling, the innermost element handles the event first, then the outer, and so on.

46. Explain the concept of memoization in JavaScript?

Memoization in JavaScript is an optimization technique that stores the results of expensive function calls and reuses them when the same inputs occur again. This reduces the number of computations by caching the results. Memoization is typically implemented using an object or a map to store function arguments and their corresponding results. When the function is called with the same arguments, the cached result is returned instead of recalculating it. This improves performance, especially for functions with repeated calls and expensive computations.

47. What is the difference between == and === in JavaScript?

In JavaScript, == is the **loose equality** operator, which compares two values for equality after performing type coercion if necessary. This means it converts the operands to the same type before comparing.

=== is the **strict equality** operator, which compares both the values and their types, without performing type conversion.

48. Explain the concept of promises and how they work.

A **Promise** in JavaScript is an object that represents the result of an asynchronous operation. It can be in one of three states: pending, fulfilled (resolved), or rejected. You create a promise using new Promise(), passing an executor function with resolve and reject callbacks. When the operation

arguments, the cached result is returned instead of recalculating it. This improves performance, especially for functions with repeated calls and expensive computations.

47. What is the difference between == and === in JavaScript?

In JavaScript, == is the **loose equality** operator, which compares two values for equality after performing type coercion if necessary. This means it converts the operands to the same type before comparing.

=== is the **strict equality** operator, which compares both the values and their types, without performing type conversion.

48. Explain the concept of promises and how they work.

A Promise in JavaScript is an object that represents the result of an asynchronous operation. It can be in one of three states: pending, fulfilled (resolved), or rejected. You create a promise using new Promise(), passing an executor function with resolve and reject callbacks. When the operation succeeds, resolve() is called; if it fails, reject() is used. Promises are handled with .then() for success and .catch() for failure. They can be chained to handle sequences of asynchronous tasks in a more readable way.

49. What is the difference between a shallow copy and a deep copy?

A shallow copy creates a new object but copies references to the original nested objects, meaning changes to the nested objects affect both the original and the copy. A **deep copy**, on the other hand, creates a new object and recursively copies all nested objects, ensuring that the original and the copy are completely independent. In a shallow copy, nested objects are shared, while in a deep copy, they are fully duplicated.

50. Explain the concept of the event loop and the call stack in JavaScript. How does JavaScript handle asynchronous code execution?

In JavaScript, the event loop manages the execution of code, handling both synchronous and asynchronous operations. The call stack stores function calls and executes them in a Last In, First Out (LIFO) order. When asynchronous code (e.g., setTimeout, promises) is encountered, it's offloaded to the callback queue once its execution context is ready. The event loop continuously checks the call stack and moves tasks from the callback queue to the stack when it's empty, allowing asynchronous code to run without blocking the main thread.

51. What are Web Workers, and how do you use them to run scripts in the background?

Web Workers are JavaScript threads that run in the background, separate from the main thread, allowing long-running scripts to be executed without blocking the user interface. You can create a Web Worker using the `Worker` constructor, passing a JavaScript file as an argument. Once created, the worker can perform tasks asynchronously, and you can communicate with it via postMessage and onmessage events, ensuring the main thread remains responsive.

52. Explain the concept of "debouncing" and "throttling" in JavaScript. How can these techniques optimize performance?

Debouncing and throttlingare techniques used to optimize performance by limiting the frequency of function executions in response to events like scrolling or resizing.

- **Debouncing** ensures that a function is only executed after a certain amount of idle time, i.e., it delays the execution until the event stops triggering for a specified time (e.g., for search input).
- **Throttling** limits the number of times a function can be executed in a given period, ensuring it runs at regular intervals (e.g., during scroll or window resizing).

JavaScript MCQ Coding Interview Questions

53. Which of the following is a JavaScript data type?

Options:

1. number
2. string
3. boolean
4. All of the above

Answer:

4

Explanation:

runs at regular intervals (e.g., during scroll or window resizing).

JavaScript MCQ Coding Interview Questions

53. Which of the following is a JavaScript data type?

Options:

1. number
2. string
3. boolean
4. All of the above

Answer:

4

Explanation:

- JavaScript has several built-in data types, and number, string, and boolean are all valid types.
- A number represents numeric values, a string represents sequences of characters, and a boolean represents either true or false. All of these are fundamental data types in JavaScript.

54. What is the result of the following code?

```
let a = [1, 2, 3];
let b = a;
b[0] = 100;
console.log(a);
```

Options:

1. [100, 2, 3]
2. [1, 2, 3]
3. [100, 100, 100]
4. undefined

Answer :

1

Explanation :

- The code snippet represents two arrays [100, 2, 3] and [1, 2, 3] being compared using the equality operator (==).
- In JavaScript, arrays are reference types, meaning each array is a reference to an object in memory.
- Since the two arrays are different objects in memory, the comparison results in false, which evaluates to undefined when logged. Therefore, the result is undefined.

55. What is the output of the following code?

```
console.log([] + []);
```

Options:

1. `null`
2. undefined
3. ''
4. []

Answer :

3

Explanation:

- The `+` operator concatenates two empty arrays, which results in an empty string `''` .

56. What will be the output of the following code?

```
(function() {
    var a = b = 5;
})();
console.log(typeof a);
console.log(typeof b);
```

Options:

1. `typeof a: "undefined"`
   `typeof b: "number"`
2. typeof a: "number"
   typeof b: "number"

- The `+` operator concatenates two empty arrays, which results in an empty string `''` .

56. What will be the output of the following code?

```
(function() {
    var a = b = 5;
})();
console.log(typeof a);
console.log(typeof b);
```

Options:

1. `typeof a: "undefined"`
   `typeof b: "number"`
2. typeof a: "number"
   typeof b: "number"
3. typeof a: "undefined"
   typeof b: "undefined"
4. typeof a: "number"
   typeof b: "undefined"

Answer:

1

Explanation:

- Inside the IIFE, `b = 5` is treated as a global variable (since no `var` , `let` , or `const` keyword is used).
- However, `a` is declared with `var` and is local to the function, so it is `undefined` outside.

57. What will be logged in the console?

```
console.log(1 < 2 < 3);
console.log(3 > 2 > 1);
```

Options:

1. `true, true`
2. `true, false`
3. `false, true`
4. `false, false`

Answer:

2

Explanation:

- `1 < 2 < 3 is evaluated as (1 < 2) < 3, which becomes true < 3. In JavaScript, true is treated as 1, so 1 < 3 is true.`
- 3 > 2 > 1 becomes (3 > 2) > 1, which results in true > 1. Since true is 1, the comparison becomes 1 > 1, which is false.

58. What will be the output of the following code?

```
const obj1 = { a: 1 };
const obj2 = { a: 1 };
console.log(obj1 == obj2);
console.log(obj1 === obj2);
```

Options:

1. `true, true`
2. `true, false`
3. `false, true`
4. `false, false`

Answer:

4

Explanation:

- In JavaScript, objects are compared by reference, not by value. Since obj1 and obj2 point to different memory locations, both == and === comparisons return false.

59. What will be the result of the following?

- 5 > 2 > 1 becomes (5 > 2) > 1, which results in true > 1. Since true is 1, the comparison becomes 1 > 1, which is false.

58. What will be the output of the following code?

```
const obj1 = { a: 1 };
const obj2 = { a: 1 };
console.log(obj1 == obj2);
console.log(obj1 === obj2);
```

Options:

1. `true, true`
2. `true, false`
3. `false, true`
4. `false, false`

Answer:

4

Explanation:

- In JavaScript, objects are compared by reference, not by value. Since obj1 and obj2 point to different memory locations, both == and === comparisons return false.

59. What will be the result of the following code?

```
let x = 10;
let y = (x++, x + 1, x * 2);
console.log(y);
```

Options :

1. 22
2. 12
3. 21
4. 20

Answer:

22

Explanation:

- The comma operator (,) evaluates all expressions but returns the value of the last one.
- x++ increments x to 11, but the result of this expression is the original 10.
- x + 1 becomes 11 + 1 = 12, and the final expression x * 2 evaluates to 11 * 2 = 22, which is assigned to y.

60. What will be the output of this asynchronous JavaScript code?

```
console.log('A');
setTimeout(() => console.log('B'), 0);
Promise.resolve().then(() => console.log('C'));
console.log('D');
```

Options:

1. A D B C
2. A B C D
3. A D C B
4. A C D B

Answer:

3

Explanation:

- The synchronous code runs first, logging 'A' and 'D'.
- Promise callbacks (microtasks) are executed before setTimeout (macrotasks). So 'C' is logged before 'B'.

61. What will be the output of this recursive function?

```
function foo(num) {
  if (num === 0) return 1;
```

before 'B'.

61. What will be the output of this recursive function?

```
function foo(num) {
  if (num === 0) return 1;
  return num + foo(num - 1);
}
console.log(foo(3));
```

Options:

1. 3
2. 6
3. 7
4. 10

Answer:

3

Explanation:

- The function works recursively:
  - foo(3) → 3 + foo(2)
  - foo(2) → 2 + foo(1)
  - foo(1) → 1 + foo(0)
  - foo(0) → 1
- So, the total is 3 + 2 + 1 + 1 = 7.

62. What will be printed in the following code?

```
let a = [1, 2, 3];
let b = a;
b.push(4);
console.log(a);
console.log(b);
```

Options:

1. `[1, 2, 3]`
   `[1, 2, 3, 4]`
2. [1, 2, 3, 4]
   [1, 2, 3, 4]
3. [1, 2, 3]
   [1, 2, 3]
4. [1, 2, 3, 4]
   [1, 2, 3]

Answer:

2

Explanation:

- In JavaScript, arrays are reference types. Both a and b point to the same array in memory. Modifying b also affects a.

63. What will be logged by the following code?

```
function test() {
  console.log(this);
}
test.call(null);
```

Options:

1. `null`
2. undefined
3. Window or global object
4. TypeError

Answer:

3

- In JavaScript, arrays are reference types. Both a and b point to the same array in memory. Modifying b also affects a.

63. What will be logged by the following code?

```
function test() {
  console.log(this);
}
test.call(null);
```

Options:

1. `null`
2. undefined
3. Window or global object
4. TypeError

Answer:

3

Explanation:

- In non-strict mode, calling a function with this set to null defaults to the global object (Window in browsers or global in Node.js).

JavaScript Interview Questions and Answers

JavaScript Tricky Coding Interview Questions

In Interviews sometimes interviewer ask some tricky output based JS questions to test your grasp on concepts. Let's see some of tricky questions that may ask in your upcoming interview.

Note: we recommend you should guess first then confirm your output by hit on run.

64.

```
let a = 5;
let b = '5';

if (a == b) {
  console.log('Equal');
} else {
  console.log('Not Equal');
}
```

Explanation: *The `==` operator performs type coercion, converting both operands to the same type before comparison. Here, `'5'` is coerced to a number, making the comparison `5 == 5`, which evaluates to `true`. To avoid such issues, it's recommended to use the strict equality operator `===`, which checks both value and type without coercion.*

65.

```
for (var i = 0; i < 3; i++) {
  setTimeout(function() {
    console.log(i);
  }, 100);
}
```

Explanation: *Due to JavaScript's function scoping with `var`, the variable `i` is shared across all iterations. By the time the `setTimeout` callbacks execute, the loop has completed, and `i` equals `3`. To capture the value of `i` at each iteration, you can use `let` (which has block scope) or pass `i` as an argument to an immediately invoked function expression (IIFE).*

66.

```
function arrayFromValue(item) {
  return
    [item];
}
```

which evaluates to `true`. To avoid such issues, it's recommended to use the strict equality operator `===`, which checks both value and type without coercion.

65.

```
for (var i = 0; i < 3; i++) {
  setTimeout(function() {
    console.log(i);
  }, 100);
}
```

Explanation: *Due to JavaScript's function scoping with `var`, the variable `i` is shared across all iterations. By the time the `setTimeout` callbacks execute, the loop has completed, and `i` equals `3`. To capture the value of `i` at each iteration, you can use `let` (which has block scope) or pass `i` as an argument to an immediately invoked function expression (IIFE).*

66.

```
function arrayFromValue(item) {
  return
    [item];
}

console.log(arrayFromValue(10)); // ???
```

Explanation: *JavaScript's automatic semicolon insertion adds a semicolon after the `return` statement, causing the function to return `undefined` instead of the intended array. To fix this, ensure the return statement is on the same line as the array:*

67.

```
const car = {
  name: 'Toyota',
  getName: function() {
    return this.name;
  },
};

const getCarName = car.getName;
console.log(getCarName()); // ???
```

Explanation: *When `getCarName` is called, it's no longer in the context of the `car` object. Therefore, `this` refers to the global object (or `undefined` in strict mode), not the `car` object. To maintain the correct context, you can use `.bind(car)`:*

const getCarName = car.getName.bind(car);
console.log(getCarName()); // 'Toyota'

68.

```
console.log(a); // ???
console.log(b); // ???

var a = 2;
let b = 2;
```

Explanation: *Variables declared with `var` are hoisted and initialized with `undefined`, so `console.log(a)` outputs `undefined`. Variables declared with `let` are hoisted but not initialized, leading to a `ReferenceError` when accessed before their declaration.*

69.

```
console.log(typeof null); // ???
```

Explanation: *This is a well-known quirk in JavaScript. Despite `null` being a primitive value representing the intentional absence of any object value, the `typeof` operator returns `"object"`. This behavior is considered a bug in JavaScript that has been maintained for backward compatibility.*

70.

```
let arr = new Array(3).fill([]);
arr[0].push(10);
console.log(arr); // ???
```

Explanation: *The `fill` method fills all elements of the array with the same reference to the same array. Therefore, when you modify one element (e.g., `arr[0].push(10)`), all elements reflect this change because they all point to the same array in memory.*

```
console.log(typeof null); // ???
```

Explanation: *This is a well-known quirk in JavaScript. Despite `null` being a primitive value representing the intentional absence of any object value, the `typeof` operator returns `"object"`. This behavior is considered a bug in JavaScript that has been maintained for backward compatibility.*

70.

```
let arr = new Array(3).fill([]);
arr[0].push(10);
console.log(arr); // ???
```

Explanation: *The `fill` method fills all elements of the array with the same reference to the same array. Therefore, when you modify one element (e.g., `arr[0].push(10)`), all elements reflect this change because they all point to the same array in memory.*

71.

```
const obj = { x: 1 };
const { x, x: y } = obj;
console.log(y); // ???
```

Explanation: *In this destructuring assignment, `x` is first assigned the value `1` from the object. Then, `x: y` creates a new variable `y` and assigns it the value of `x`. Since `x` is `1`, `y` also becomes `1`. This demonstrates how destructuring can assign values to multiple variables from the same property.*

72.

```
console.log(typeof undeclaredVar); // ???
console.log(typeof y); // ???
let y = 1;
```

Explanation: *The first `console.log` outputs `"undefined"` because `undeclaredVar` is not declared at all. The second `console.log` throws a `ReferenceError` because `y` is declared with `let` and is in a "temporal dead zone" from the start of the block until the declaration is encountered. During this period, accessing `y` results in a `ReferenceError`.*

Common JavaScript Interview Questions

1. What are the different data types in JavaScript?

JavaScript has two types of data: **primitive** and **non-primitive**.

- **Primitive data types**: `string`, `number`, `boolean`, `undefined`, `null`, `symbol`, `bigint`
- **Non-Primitive data types**: `objects`, arrays and functions.

2. What is the difference between == and === in JavaScript?

- **== (loose equality):** They compares only the values, allowing type coercion (i.e., converts types if necessary).
- **=== (strict equality):** They compares both value and type, without type conversion.

3. What is the difference between `let`, `const`, and `var`?

- `var`: Function-scoped, can be re-assigned and redeclared within its scope.
- `let`: Block-scoped, can be reassigned but not redeclared within the same scope.
- `const`: Block-scoped, cannot be reassigned or redeclared, and the value assigned to it remains constant.

4. Explain hoisting in JavaScript.

Hoisting is JavaScript's default behavior of moving all variable and function declarations to the top of their containing scope during the compile phase. However, only the declarations are hoisted, not the initialization.

5. What is the difference between `null` and `undefined` in JavaScript?

- `null`: Represents an intentional absence of any object value. It is explicitly assigned to indicate "no value."
- **`undefined`**: Indicates that a variable has been declared but has not yet been assigned a value.

```
let a = null;    // Explicitly assigned nulllet b;          // Variable
declared but not assigned, hence undefi
```

constant.

4. Explain hoisting in JavaScript.

Hoisting is JavaScript's default behavior of moving all variable and function declarations to the top of their containing scope during the compile phase. However, only the declarations are hoisted, not the initialization.

5. What is the difference between `null` and `undefined` in JavaScript?

- `null`: Represents an intentional absence of any object value. It is explicitly assigned to indicate "no value."
- **`undefined`**: Indicates that a variable has been declared but has not yet been assigned a value.

```
let a = null;    // Explicitly assigned nulllet b;          // Variable
declared but not assigned, hence undefined
```

6. What are promises in JavaScript and how do they work?

A promise is an object representing the eventual completion or failure of an asynchronous operation. Promises are used to handle asynchronous operations like API calls, ensuring cleaner code compared to callbacks. It has three states:

- **`pending`:** The initial state.
- **`fulfilled`:** The operation completed successfully.
- **`rejected`:** The operation failed.

```
const myPromise = new Promise((resolve, reject) => {
 let success = true;
if (success) {
 resolve("Operation was successful");
 }
 else {
    reject("Operation failed");
}});
myPromise.then(result => console.log(result)).catch(error =>
console.log(error));
```

7. What is the event loop in JavaScript?

The event loop is a mechanism that allows JavaScript to handle asynchronous operations (like I/O, timers, etc.) without blocking the main thread. It continuously checks the call stack for any code to execute and moves tasks from the callback queue to the call stack when the stack is empty.

8. What are closures in JavaScript?

A closure is a function that retains access to its lexical scope (the scope in which it was created) even after that scope has finished execution. Closures allow functions to access variables from an outer function after the outer function has returned.

```
function outer() {
    let x = 10;
    return function inner() {
        console.log(x);
    }
}
const closureFunc = outer();
closureFunc(); // prints 10
```

Output

```
10
```

9. Explain the concept of `this` in JavaScript.

In JavaScript, the `this` keyword refers to the context in which a function is called. It is used to refer to the object that is executing the current piece of code.

The value of `this` can change depending on how the function is called. Here are the different scenarios where `this` behaves differently:

- **Global context**: In non-strict mode, `this` refers to the global object (`window` in browsers).
- **Object method**: `this` refers to the object the method belongs to.
- **Constructor function**: this refers to the instance of the object being created.
- **Arrow functions**: In arrow functions, `this` is lexically bound to the surrounding context.

Output

```
10
```

9. Explain the concept of `this` in JavaScript.

In JavaScript, the `this` keyword refers to the context in which a function is called. It is used to refer to the object that is executing the current piece of code.

The value of `this` can change depending on how the function is called. Here are the different scenarios where `this` behaves differently:

- **Global context**: In non-strict mode, `this` refers to the global object (`window` in browsers).
- **Object method**: `this` refers to the object the method belongs to.
- **Constructor function**: `this` refers to the instance of the object being created.
- **Arrow functions**: In arrow functions, `this` is lexically bound to the surrounding context.

10. What is a callback function in JavaScript?

A **callback function** in JavaScript is a function that is passed as an argument to another function and is executed after the completion of that function. Callback functions are primarily used for handling asynchronous operations, such as API requests or timeouts, ensuring that certain code runs only after a specific task is completed.

```
function greet(name, callback) {
    console.log("Hello " + name);
    callback();
}

function sayGoodbye() {
    console.log("Goodbye!");
}

greet("Anjali", sayGoodbye);
```

Output

```
Hello Anjali
Goodbye!
```

Pratice JavaScript with Quizes

Apart from these questions you can also practice JavaScript Quiz for better understanding of every topic to enhance your knowledge and helping you in the interviews.

1. Basic JavaScript
2. Variables and Data Types
3. Operators
4. Control Flow
5. Functions
6. Objects
7. Arrays
8. DOM and BOM
9. Event Handling
10. Classes and Inheritance
11. Modern JavaScript (ES6+)
12. Advanced JavaScript
13. Regular Expressions and JSON
14. Asynchronous JavaScript
15. Error Handling and Debugging

Conclusion

In conclusion, after going through this article, you should now have a solid understanding of the key JavaScript interview questions that are commonly asked in web development interviews. Remember, JavaScript often includes a few tricky questions, so be sure not to skip that part of your preparation. Whether you're preparing for a front-end or back-end role, it's always helpful to check out our specialized interview question guides(1. Frontend Interview Questions 2. Backend Interview Questions 3. Full-Stack Interview Questions) for both areas. Stay well-prepared and confident, and you'll be ready to tackle your interview with ease!