JS INTERVIEW Q&A

1. Difference between " == " and " === " operators.

1. Both are comparison operators and they return boolean values always. The difference between both the operators is that "==" is used to compare values whereas " === " is used to compare both value and datatype.

2. var x="2";
   var y=2;

3. console.log(x==y); It returns true because it only check values not data types. It is able to check the values because in == it will do type coercion to change the data type so that the value can be compared.

4. console.log(x===y); It returns false because you can see that values are same but x is of String type and y is of Number type. You can find the type of any value by using typeof operator.

5. So in strict equal comparison operator both value and data types should be same like x=2 and y=2 otherwise it returns false.

****

2. What is a spread operator?

The JavaScript spread operator (...) allows us to destructure the non-primitive data types like arrays and objects to access the elements individually.

Code:

```
const numbersOne = [1, 2, 3];
const numbersTwo = [4, 5, 6];
const numbersCombined = [...numbersOne, ...numbersTwo];
console.log(numbersCombined)
```

Output:

[1,2,3,4,5,6]

****

3. What are the differences between var, let and const?

var is a global scoped means variables created using var keyword will be accessible globally. It can be redeclared and reinitialized.

```
var a = 10;
```

let is a block scoped means variables created using let keyword will be accessible only in that specific block where it has been declared. It cannot be redeclared but it can be reinitialized.

```
let a = 20;
```

const is also block scoped means variables created using const keyword will be accessible only in that specific block where it has been declared. It cannot be redeclared and cannot be reinitialized.

```
const a = 30;
```

****

4. What is execution context?

Each and every browser is having a JavaScript Engine, example Google is having V8 engine, Internet Explorer is having Chakra and Mozilla Firefox is having Spider Monkey.

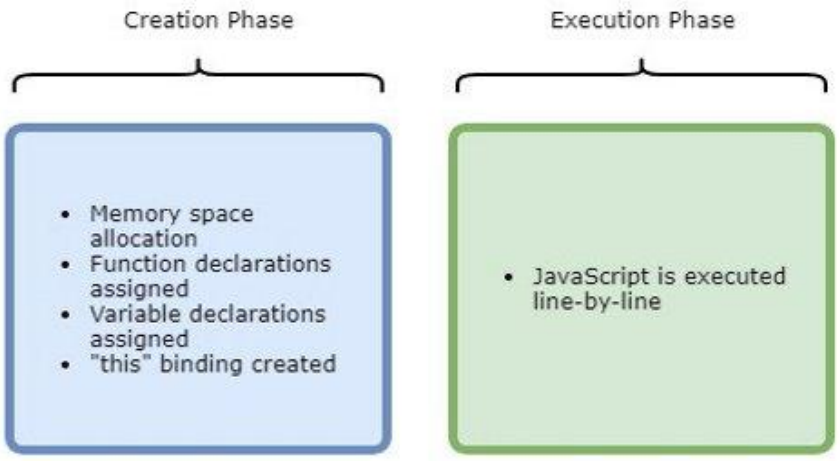
JS engine uses a Call stack and also creates a special environment to handle the execution of the JavaScript code. This environment is known as the Execution Context.

The Execution Context contains the code that's currently running, and everything that aids in its execution.

During the Execution Context run-time, the specific code gets parsed by a parser, the variables and functions are stored in memory, executable byte-code gets generated, and the code gets executed.

There are two kinds of Execution Context in JavaScript:

- Global Execution Context (GEC)
- Function Execution Context (FEC)



The moment code gets executed GEC gets created and gets pushed to Call Stack. Inside the global or functional execution context, there are two phases : Memory allocation (Creation Phase) and Code execution (Execution Phase).

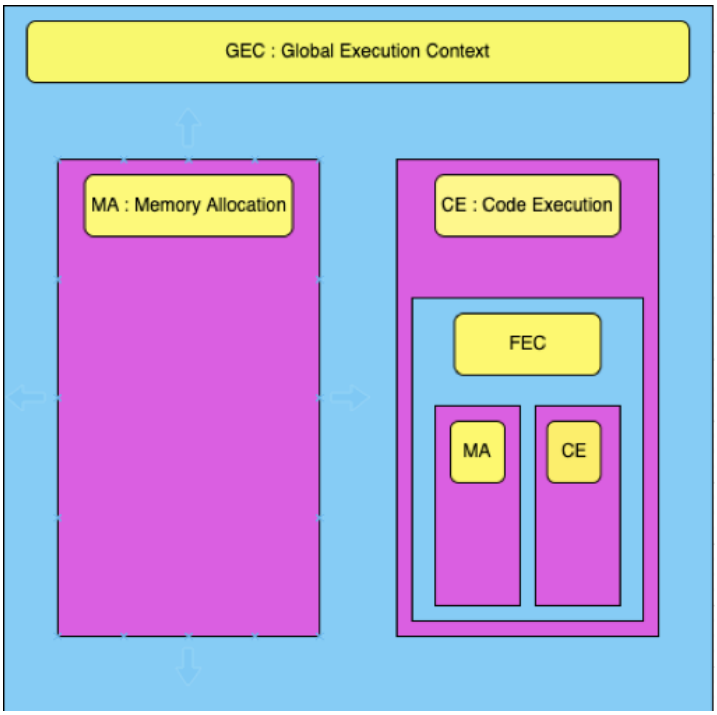
Inside Memory allocation, the variables declared using var keyword get memory allocated as undefined in global scope and variables declared using let or const gets their memory allocated as undefined in block scope.

That is the reason why we can access the variables before initialization declared using var, it will give undefined.

But if we will try to access the variables using let const before their initialization then it will give Reference Error because these variables will be in block scope and can not be accessed until unless they are not initialized. This is also known as temporal dead zone.

Whereas, functions gets memory allocated as their actual value defined in code. Inside Code Execution, variables will get their actual value assigned to them. And functions will create a functional execution context.

The moment this FEC will be created, it will be pushed to Call stack and once the execution of the function is over then this FEC will be removed from the call stack and the point of execution will again be moved to GEC.



****

5. What is meant by first class functions

A programming language is said to have First-class functions when functions in that language are treated like any other variable. For example, in such a language, a function can be passed as

an argument to other functions, can be returned by another function and can be assigned as a value to a variable.

1. Example | Assign a function to a variable

```
const temp = function() {
  console.log("Hello World !!");
}
temp();
```

Output: Hello World !!

We assigned an Anonymous Function in a Variable, then we used that variable to invoke the function by adding parentheses () at the end.

2. Example | Pass a function as an Argument

```
function sayHello() {
  return "Hello,  ";
}
function greeting(helloMessage, name) {
 console.log(helloMessage() + name);
}
// Pass “sayHello” function as an argument to “greeting” function
greeting(sayHello, "JavaScript!");
```

Output: Hello, JavaScript!

3. Example | Return a function

```
function sayHello() {
  return function add() {
    console.log("Hello!");
  }
}
```

In this example; We need to return “add” function from another function so add function will be first class function. We can return a function because we treated function in JavaScript as a value.

****

5.What are closures?

A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

```
function outerFunc() {
 var outerVar = 100;
 function innerFunc() {
    console.log(outerVar);
 }
 return innerFunc;
}
 var value=outerFunc();
 value();
```

Output: 100

****

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

These all three methods are used to invoke a function where we are supposed to pass an object as first argument and at the time of definition we don't have mention this object as a parameter and we can access the values of object by using this keyword in function definition.

call(): The call() method invokes a function in which first argument will be the object and rest of the arguments required by function will be provided as an individual arguments.

```
var emp1={
  name:"John",
  age:21
}
var emp2={
  name:"Smith",
  age:22
}
function invite(greeting1,greeting2){
        console.log(greeting1+" "+this.name+" "+greeting2)
}
invite.call(emp1,"Hi","How are you!")
```

Output: Hi John How are you!

apply(): The apply() method invokes a function in which first argument will be the object and rest of the arguments will be passed as an array of elements.

```
invite.apply(emp2,["hey dude," ,"how are you!"])
```

Output: hey dude, Smith how are you!

bind: The bind() method returns a new function and this function will be having the reference of the object passed, now whenever you want to use this returned function in the code you can use it by passing rest of the arguments.

```
var inviteEmployee1=invite.bind(emp1);
var inviteEmployee2=invite.bind(emp2);
inviteEmploye1("hi","how are you")
inviteEmploye2("hey","how are you doing?")
```

Output:

```
hi John how are you
hey Smith how are you doing?
```

****

7. Explain prototypes

Prototypes are the mechanism by which JavaScript objects inherit features from one another.

Every object in JavaScript has a built-in property, which is called its prototype.

The prototype is itself an object, so the prototype will have its own prototype, making what's called a prototype chain.

The chain ends when we reach a prototype that has null for its own prototype.

The JavaScript prototype property allows you to add new properties to object constructors:

```
function Person(first, last, age, eyecolor) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eyecolor;
}
```

```
Person.prototype.nationality = "English";
const myFather = new Person("John", "Doe", 50, "blue");
console.log("The nationality of my father is " + myFather.nationality;)
```

****

8.What are promises and why do we need them?

Promises are used to handle asynchronous operations in JavaScript. They are easy to manage when dealing with multiple asynchronous operations where callbacks can create callback hell leading to unmanageable code.

Multiple callback functions would create callback hell that leads to unmanageable code. Also it is not easy for any user to handle multiple callbacks at the same time.

Events were not good at handling asynchronous operations.

Promises are the ideal choice for handling asynchronous operations in the simplest manner. They can handle multiple asynchronous operations easily and provide better error handling than callbacks and events.

A Promise is in one of these states:

pending: initial state, neither fulfilled nor rejected.

fulfilled: meaning that the operation was completed successfully.

rejected: meaning that the operation failed.

Benefits of Promises

1. Improves Code Readability
2. Better handling of asynchronous operations
3. Better flow of control definition in asynchronous logic

4. Better Error Handling

```
var promise = new Promise(function(resolve, reject) {
        const x = "prepbytes";
        const y = "prepbytes"
        if(x === y) {
                resolve();
        } else {
                reject();
        }
});

promise.
        then(function () {
                console.log('Success,you are a prepbytes student');
        }).
        catch(function () {
                console.log('Some error has occurred');
        });
```

Promise Consumers

Promises can be consumed by registering functions using .then and .catch methods.

1. then()

then() is invoked when a promise is either resolved or rejected. It may also be defined as a career which takes data from promise and further executes it successfully.

Parameters:

then() method takes two functions as parameters.

1)First function is executed if promise is resolved and a result is received.

2)Second function is executed if promise is rejected and an error is received. (It is optional and there is a better way to handle error using .catch() method

```
.then(function(result){
      //handle success
   }, function(error){
      //handle error
   })
```

```
var promise = new Promise(function(resolve, reject) {
        resolve('prepbytes');
})
```

```
promise
        .then(function(successMessage) {
        //success handler function is invoked
                console.log(successMessage);
        }, function(errorMessage) {
                console.log(errorMessage);
        })
```

2. catch()

catch() is invoked when a promise is either rejected or some error has occurred in execution. It is used as an Error Handler whenever at any step there is a chance of getting an error.

Parameters:

catch() method takes one function as parameter.

Function to handle errors or promise rejections.(.catch() method internally calls .then(null, errorHandler), i.e. .catch() is just a shorthand for .then(null, errorHandler))

```
.catch(function(error){
      //handle error
   })

var promise = new Promise(function(resolve, reject) {
        reject('Promise Rejected')
})

promise
        .then(function(successMessage) {
                console.log(successMessage);
        })
        .catch(function(errorMessage) {
        //error handler function is invoked
                console.log(errorMessage);
        });
```

Applications

1)Promises are used for asynchronous handling of events.

2)Promises are used to handle asynchronous http requests.

****

9.What is the purpose of async/await keywords?

An async function is a function declared with the async keyword, and the await keyword is permitted within it. The async and await keywords enable asynchronous, promise-based behavior to be written in a cleaner style, avoiding the need promise chains.

The use of await pauses the async function until the promise returns a result (resolve or reject) value. For example,

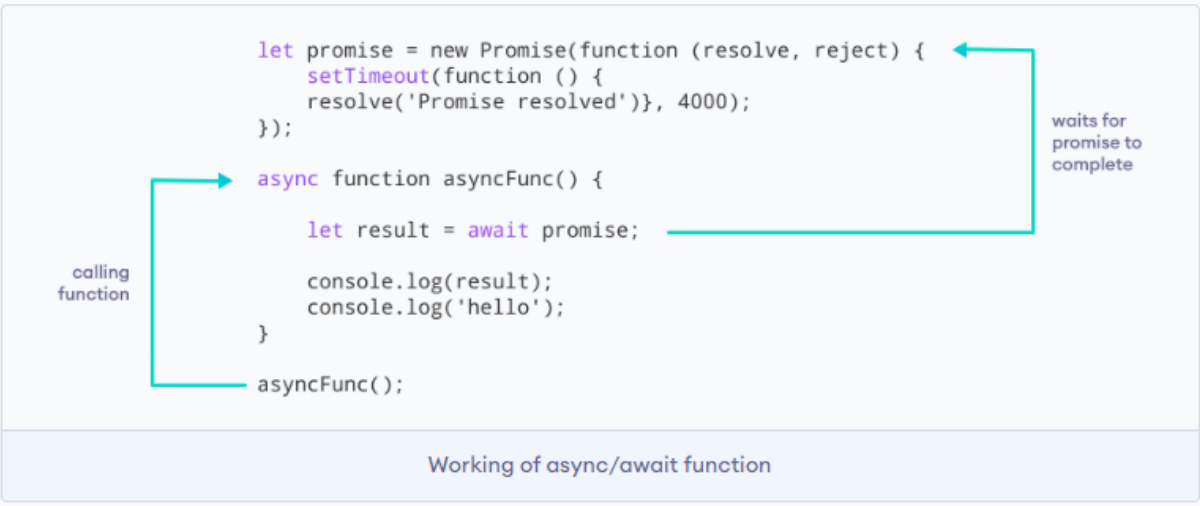
```
let promise = new Promise(function (resolve, reject) {
    setTimeout(function () {
    resolve('Promise resolved')}, 4000);
});


async function asyncFunc() {

    // wait until the promise resolves
    let result = await promise;

    console.log(result);
    console.log('hello');
}

// calling the async function
asyncFunc();
```



Working of async/await function

****

10.What are constructor functions in JS?

A constructor is a special function that creates and initializes an object instance of a class. In JavaScript, a constructor gets called when an object is created using the new keyword. We have multiple constructor functions like Function constructor for functions, Object constructor function for objects, Array constructor function for arrays.

When a constructor gets invoked in JavaScript, the following sequence of operations take place:

- A new empty object gets created.
- The this keyword begins to refer to the new object and it becomes the current instance object.
- The new object is then returned as the return value of the constructor.

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

var user1 = new User('Bob', 25);
var user2 = new User('Alice', 27);
```

****

11.What is hoisting?

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution. Remember that JavaScript only hoists declarations, not initialisation. Let's take a simple example of variable hoisting. Let and const keywords are not hoisted so when you try to access them before initialization they start giving you Reference error.

```
console.log(message); //output : undefined
var message = 'The variable Has been hoisted';
```

The above code looks like as below to the interpreter that is why it gives us undefined.

```
var message;
console.log(message);
message = 'The variable Has been hoisted';
```

****

12. What is the DOM?

The Document Object Model (DOM) is the data representation of the objects that comprise the structure and content of a document on the web.

The Document Object Model (DOM) is a programming interface for web documents. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as nodes and objects; that way, programming languages can interact with the page.

****

13.Difference between undefined vs not defined vs NaN

Undefined

undefined is a global variable that JavaScript creates at run time. JavaScript assigns undefined to any variable that has been declared but not initialized. In other words, in a case where no value has been explicitly assigned to the variable, JavaScript calls it undefined.

null

It is one of JavaScript's primitive data type and is treated as falsy for boolean operations.null is an empty or non-existent value and null must be assigned. We use null when we want to explicitly declare that a variable is empty.

not defined

A not defined is a variable that is not declared inside the code at a given point of time with declaration keyword like var, let, or const.

****

14.How many operators do we have in JS ?

We have 6 types of operators in JS they are

1. Arithmetic Operators : + - * / %
2. Comparison Operators : == === != !==
3. Logical Operators : && || !
4. Assignment Operators : =
5. Conditional Operators
6. Ternary Operator ?:

****

15.What are pure functions?

Pure Function is a function (a block of code) that always returns the same result if the same arguments are passed. It does not depend on any state, or data change during a program's execution rather it only depends on its input arguments.

Let's see the below JavaScript Function:

```
function calculateGST( productPrice ) {
   return productPrice * 0.05;
}
```

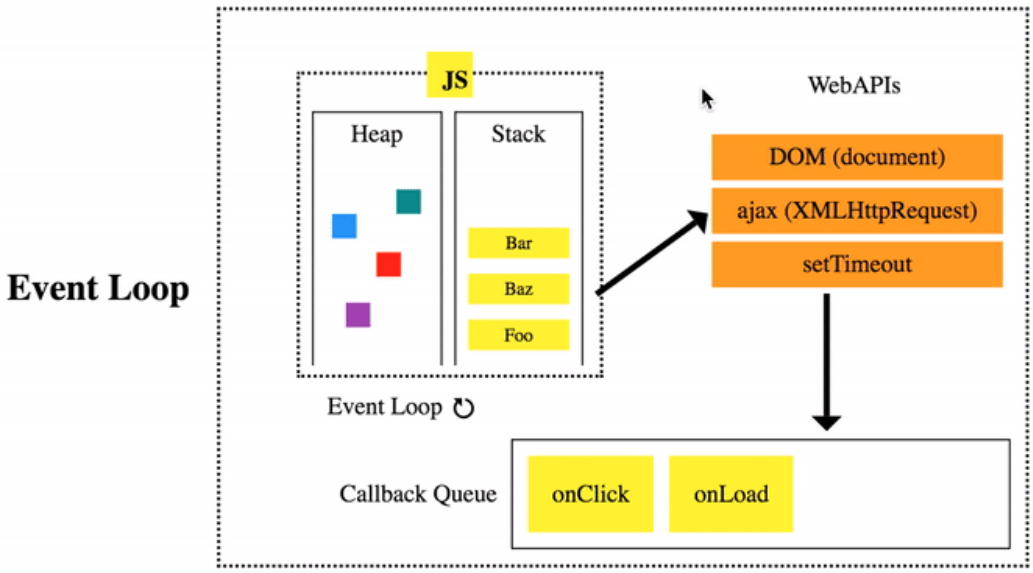
The above function will always return the same result, if we pass the same productPrice. In other words, it's output doesn't get effected by any other values / state changes. So we can call "calculateGST" function as a Pure function.

****

16.What is an event loop and call stack

A call stack is a mechanism for an interpreter (like the JavaScript interpreter in a web browser) to keep track of its place in a script that calls multiple functions — what function is currently being run and what functions are called from within that function, etc. or we can say that it stores global execution context and function execution context.

The event loop is a constantly running process that monitors both the callback queue and the call stack. All the web apis functions like setTimeout, setInterval, fetch calls etc. will not be directly executed inside the execution context, firstly they will be moved to callback queue and then the moment call stack gets empty these functions will be pushed to call stack from callback queue by event loop.



If the call stack is not empty, the event loop waits until it is empty and places the next function from the callback queue to the call stack. If the callback queue is empty, nothing will happen

****

17. What is prototype chain

Prototype chaining is used to build new types of objects based on existing ones. It is similar to inheritance in a class based language.

The prototype on object instance is available through Object.getPrototypeOf(object) or proto property whereas prototype on constructors function is available through Object.prototype.

Suppose we have an array fruits

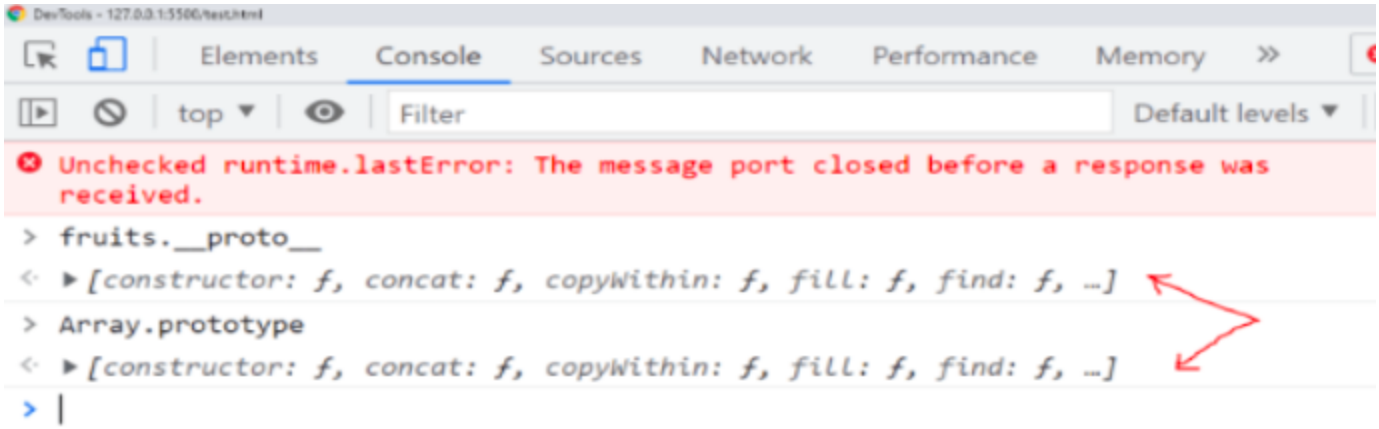
let fruits=["Apple", "Orange"];

Now we can use some methods such as concat(),index0f(),join() etc to the fruits array to obtain certain results. But from where these methods came?.

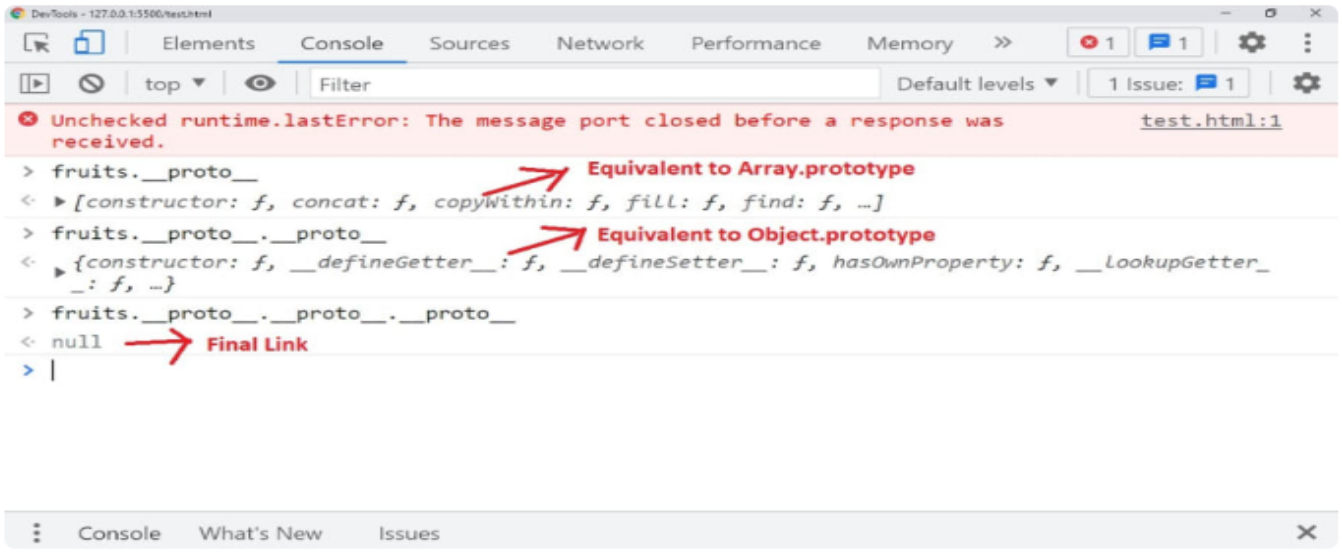
They came through the prototype of the array. i.e:fruits hold a link to array objects. You can view it in the console by typing Array.prototype in the console.

To know how they link with each other you can check by using *__proto__* in the console which is a getter function that exposes the prototype of an object. i.e fruits. *__proto__*

You will see that fruits.*__proto__* values are equivalent to Array.prototype.



fruits array is linked to Array Prototype by using the *__proto__* getter function in the console. So, in the similar way Array Prototype also has a prototype called Object prototype and if we try to view the prototype object by using *__proto__* to Object prototype then it will reach the final link as null prototype.



****

18.What is the use of setTimeout?

The setTimeout() is a method inside the window object, it calls the specified function or evaluates a JavaScript expression provided as a string after a given time period for only once.

We all have used alarms or reminders several times, this setTimeout() method also has the same purpose in web applications. We use this to delay some kind of executions.

```
Syntax: setTimeout(()=>{
        //code
}, time);
```

****

19. What is callback hell?

Callback Hell is a pattern with multiple nested callbacks which makes code hard to read and debug when dealing with asynchronous logic. The callback hell looks like a pyramid structure.

```
async1(function(){
   async2(function(){
      async3(function(){
         async4(function(){
            ....
         });
      });
   });
});
```

****

20.What is promise chaining

The process of executing a sequence of asynchronous tasks one after another using promises is known as Promise chaining. Let's take an example of promise chaining for calculating the final result,

```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000);

}).then(function(result) {

  console.log(result); // 1
  return result * 2;

}).then(function(result) {

  console.log(result); // 2
  return result * 3;

}).then(function(result) {

  console.log(result); // 6
  return result * 4;

});
```

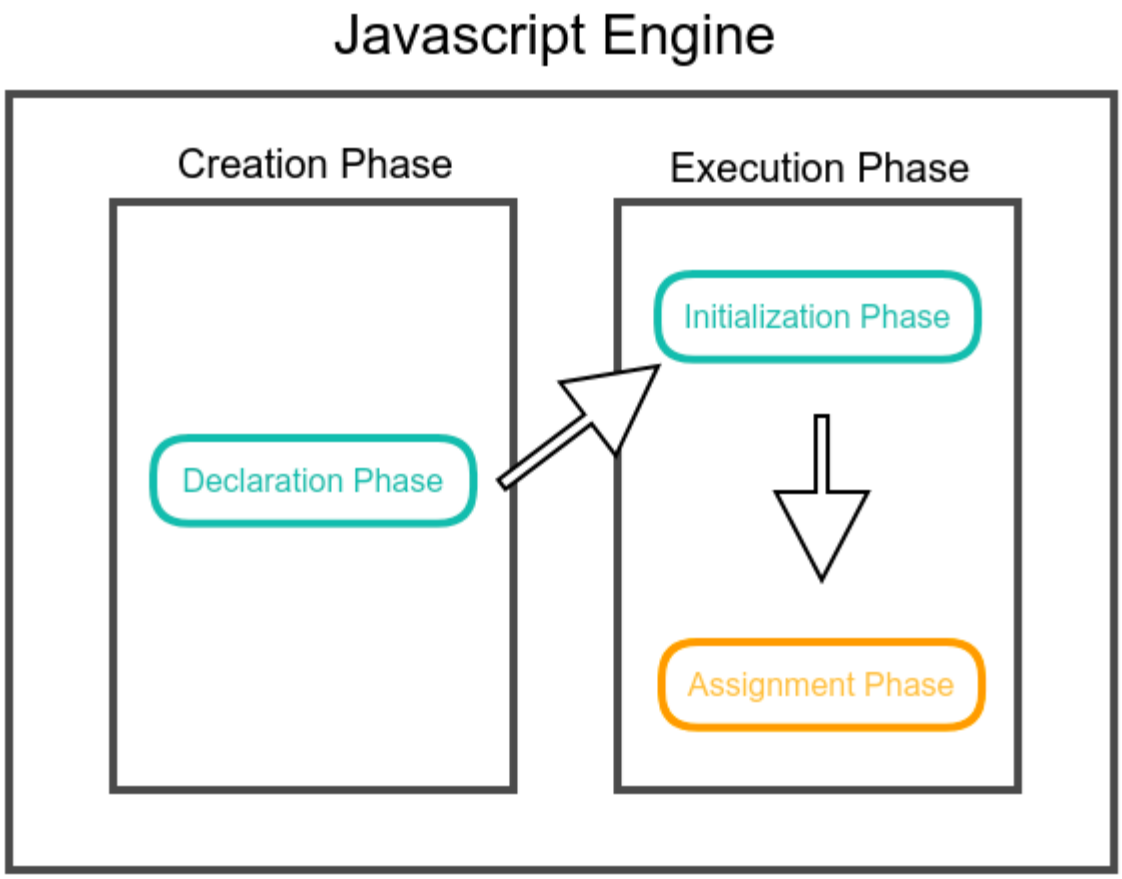
In the above handlers, the result is passed to the chain of .then() handlers with the below work flow

1. The initial promise resolves in 1 second,
2. After that .then handler is called by logging the result(1) and then return a promise with the value of result * 2.
3. After that the value passed to the next .then handler by logging the result(2) and return a promise with result * 3.
4. Finally the value passed to the last .then handler by logging the result(6) and return a promise with result * 4.

****

21.What is creation phase and execution phase?

Compiler runs through the entire code for 2 time before actually executing the code, and generates a Global Execution context , inside this we have two phases creation phase and execution phase.



Creation Phase

1. In the first run, It picks all function declarations and stores them in memory with their reference.
2. In the second run, It picks all variables and assign undefined to them. In the event of a conflict between variable and function declaration name then that variable is ignored.

Execution Phase

1. Variables assigned with values given in code at the time of initialization.
2. Functions executed

****

22.What is a Temporal Dead Zone?

The Temporal Dead Zone is a behavior in JavaScript that occurs when declaring a variable with the let and const keywords, but not with var. In ECMAScript 6, accessing a let or const variable before its declaration (within its scope) causes a ReferenceError. The time span when that happens, between the creation of a variable's binding and its declaration, is called the temporal dead zone.

Let's see this behavior with an example,

```
function somemethod() {
  console.log(counter1); // undefined
  console.log(counter2); // ReferenceError
  var counter1 = 1;
  let counter2 = 2;
}
```

****

23.What is the for-in loop in JavaScript? Give its syntax

The for-in loop is a special type of a loop that iterates over the properties of an object, or the elements of an array. for - in loop : returns you the indexes of the array. The generic syntax of the for-in loop is:

```
for(variable in object) {
   // Code to be executed
}
```

****

24.What are arrow functions?

Arrow Functions — also called “fat arrow” functions, are relatively a new way of writing concise functions in JavaScript. They have been introduced by the ECMAScript 6 specifications and since then become the most popular ES6 feature. Arrow functions allow us to use the fat arrow => operator to quickly define JavaScript functions, with or without parameters. We are able to omit the curly braces and the function and return keywords.

```
// function expression
let x = function(x, y) {
  return x * y;
}
```

can be written as

```
// using arrow functions
let x = (x, y) => x * y;
```

****

25.What are objects in javascript?

In JavaScript, an object is an unordered collection of key-value pairs. Each key-value pair is called a property.

The key of a property can be a string. And the value of a property can be any value, e.g., a string, a number, an array, and even a function.

JavaScript provides you with many ways to create an object. The most commonly used one is to use the object literal notation.

The following example creates an empty object using the object literal notation:

```
let empty = {};
```

To create an object with properties, you use the key:value within the curly braces. For example, the following creates a new person object:

```
let person = {
    firstName: 'John',
    lastName: 'Doe'
};
```

****

26.What are callbacks?

A callback function is a function passed into another function as an argument. This function is invoked inside the outer function to complete an action. Let's take a simple example of how to use callback function

```
function callbackFunction(name) {
  console.log('Hello ' + name);
}

function outerFunction(callback) {
  let name = prompt('Please enter your name.');
  callback(name);
}

outerFunction(callbackFunction);
```

****

27. Explain Local Scope, Block Scope, Functional Scope and Scope Chain in javascript

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

There are three types of scopes in JS:

1. Global Scope
2. Local or Function Scope
3. Block Scope

Global scope:A variable declared at the top of a program or outside of a function is considered a global scope variable.

```
var a=3;

function greet()
{
console.log(a);
}
greet()
```

Local scope:A variable can also have a local scope, i.e it can only be accessed within a function.

```
var a=4;
function add()
{
var  q=4;
console.log(a+b);
}
greet()
```

```
 console.log(a+q);//error
```

q will be not accesible as it was defined inside the function

Block scope: It 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 abcd = 45;
}
 console.log(abcd);reference error as it was defined inside the block
```

scope chain: JavaScript engine also uses Scope to find variables.

```
var x=24;
function OuterFunction()
{
var s=224;
var innerfunction=function innerfunction()
{
console.log(s);
}
var innerfunction2=function innerfunction2()
{
console.log(x);
}
innerfunction();
innerfunction2();
}
OuterFunction();
```

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, reference error is thrown.

****

28. What is difference between null and undefined and where to use what?

In JavaScript, undefined means a variable has been declared but has not yet been assigned a value, such as:

```
var testVar;
alert(testVar); //shows undefined

alert(typeof testVar); //shows undefined
```

null is an assignment value. It can be assigned to a variable as a representation of no value:

```
var testVar = null; alert(testVar); //shows null
 alert(typeof testVar); //shows object
```

****

29. Explain if JS is Synchornous and Asynchronous ?

By default, JavaScript is a synchronous, blocking, single-threaded language. This simply means only one operation will be in progress at a time. Windows alert function is a good example like,alert("Hello World") when this function is executed whole web page is stuck and you can't interact with it until you dismiss the alert.

****

Implementation Normal

1.Give an example of closure

```
function makeFunc() {
  var name = 'Mozilla';
  function displayName() {
    alert(name);
  }
  return displayName;
}

var myFunc = makeFunc();
myFunc();
```

****

2. Give an example of call(), apply(), bind()

```
var employee1 = {firstName: 'John', lastName: 'Rodson'};
var employee2 = {firstName: 'Jimmy', lastName: 'Baily'};

function invite(greeting1, greeting2) {
   console.log(greeting1 + ' ' + this.firstName + ' ' + this.lastName+ ', '+ greeting2);
}


invite.call(employee1, 'Hello', 'How are you?'); // Hello John Rodson, How are you?
invite.call(employee2, 'Hello', 'How are you?'); // Hello Jimmy Baily, How are you?

invite.apply(employee1, ['Hello', 'How are you?']); // Hello John Rodson, How are you?
invite.apply(employee2, ['Hello', 'How are you?']); // Hello Jimmy Baily, How are you?

var inviteEmployee1 = invite.bind(employee1);
var inviteEmployee2 = invite.bind(employee2);
inviteEmployee1('Hello', 'How are you?'); // Hello John Rodson, How are you?
inviteEmployee2('Hello', 'How are you?'); // Hello Jimmy Baily, How are you?
```

****

3. Write code to explain map and filter in arrays

```
const numbers = [1, 2, 3, 4];
const doubled = numbers.map(item => item * 2);
console.log(doubled); // [2, 4, 6, 8]
```

```
var numbers = [1,2,3,4,5];
var greaterThan2 = numbers.filter(n => n > 2);
greaterThan2; // [3,4,5]
```

****

4. **Give an example of async/await**

```
function resolveAfter2Seconds()
 return new Promise(resolve =>
 {
  setTimeout(() => {
 resolve('resolved')
 }, 2000);
  });
}
async function asyncCall() {
 console.log('calling');
  const result = await resolveAfter2Seconds();
console.log(result);
// expected output: "resolved"
```

****

5.Give an example of inheritance using function constructor

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

function Developer(name, age, gender, id, specialization) {

    // Calling Employee constructor function
    Employee.call(this, name, age, gender, id);

    // Adding a new parameter
    this.specialization = specialization;
}

// Creating objects
let Employee1 = new Employee("Suraj", 28, "Male", 564);
let Developer1 = new Developer("Karishma", 31, "Female", 345,
    "Frontend Developer");
console.log(Employee1);
console.log(Developer1);
```

Output:

```
Employee {name: 'Suraj', age: 28, gender: 'Male', id: 564}
age: 28
gender: "Male"
id: 564
name: "Suraj"
[[Prototype]]: Object

Developer {name: 'Karishma', age: 31, gender: 'Female', id: 345,
    specialization: 'Frontend Developer'}
age: 31
gender: "Female"
id: 345
name: "Karishma"
specialization: "Frontend Developer"
[[Prototype]]: Object
```

We can observe that the constructor function of Employee is inherited to create a new constructor function Developer which can be used to create objects with new properties along with the inherited properties of the parent constructor.

****

6. **Please explain Self Invoking Function and its code ?**

IIFE (Immediately Invoked Function Expression) is a JavaScript function that runs as soon as it is defined and it is anonymous function. The signature of it would be as below,

```
(function ()
   // logic here
   }
 )();
```

The primary reason to use an IIFE is to obtain data privacy because any variables declared within the IIFE cannot be accessed by the outside world. i.e, If you try to access variables with IIFE then it throws an error as below,

```
(function ()
       {
        var message = "IIFE";
        console.log(message);      //IIFE
       }
 )();
console.log(message); //Error: message is not defined
```

****

DOM

1.Create a button , on click of which it adds a new Heading tag h1 with text as "MERN stack".

```
<div class=”temp”></div>
<button onClick=”change_text()”></button>


function change_text(){
        const heading = document.createElement(“h1”);
        heading.setAttribute(“class”,”heading”);
        document.getElemenetById(“temp”).appendChild(heading);
}
```

****

2.Write code to get 1st H1 element from a webpage using DOM

```
var result=document.getElementsByTagName("h1")[0];
```

****

3.Write code to implement timer clock using JS -- display HH:MM:SS -- the time should keep updating every second

```
function myfunction(){
var d=new Date().toLocaleTimeString();
document.getElementById("demo").innerHTML=d;
}
var oneSecond = 1000;
setInterval( myfunction, oneSecond);
```

****

4.Create an HTML page having content as Hello world and a button named Replace Text. When user will click on this button page content should be changed to "Welcome to Elevation academy"

```
function getOption(){
        document.getElementById("p1").innerHTML = "Elevation Academy"
}
```

****

5. **Create an HTML page having content as Hello world and a button named "Hide Text." When user will click on this button hide the "Hello World" text**

```
function getOption(){
```

```
        document.getElementById("p1").style.display="none"
    }
```

****

Coding Question

1. **Given an array of 0's and 1's find out number of 0's**

```
function numberOfZeros(array) {
   var zeros = [];
   var ones=[]
   for(var i = 0; i < array.length; i ++){
      if(array[i] === 0) {
         zeros.push(array[i]);
      }
   }
   console.log("Numbers of zeros ="+zeros.length);
   for(var j = 0; j < array.length; j ++){
  if(array[j] === 1) {
         ones.push(array[j]);
      }
   }
   console.log("Numbers of ones ="+ones.length);
}
```

```
numberOfZeros([4, 0, 1, 5, 0]);
```

****

2. Given an array find out total no. of odd and even nos.

```
const array = [1, 2, 3, 4, 5];
let odd = 0;
let even = 0;
  for (let i = 0; i < array.length; i++) {
      // checking if a number is completely
      // divisible by 2
      if (array[i] & 1 == 1)
          odd++;
      else
          even++;
  }
console.log("Even =", even);
console.log("Odd = ", odd);
```

****

3. Given a string find out number of vowels

```
const countVowels = str => Array.from(str) .filter(letter => 'aeiou'.includes(letter)).length;
console.log(countVowels('abcdefghijklmnopqrstuvwxyz')); // 5
console.log(countVowels('test')); // 1 console.log(countVowels('ddd')); // 0
```

****