**Javascript interview questions**

1. **Difference between == and ===**

* == does the type conversion of values before comparison
* === compares value as well as type of given values
* == comparison rules

1. If one is null or undefined other operand should also be null or undefined. Otherwise returns false
2. If one of the operands is an object and the other is a primitive [convert the object to a primitive](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#primitive_coercion).
3. If the operands have the same type, they are compared as follows:
   1. Object: return true only if both operands reference the same object.
   2. String: return true only if both operands have the same characters in the same order.
   3. Number: return true only if both operands have the same value. +0 and -0 are treated as the same value. If either operand is NaN, return false; so NaN is never equal to NaN.
   4. Boolean: return true only if operands are both true or both false.
   5. BigInt: return true only if both operands have the same value.
   6. Symbol: return true only if both operands reference the same symbol.
   7. If one of the operands is a Boolean but the other is not, [convert the boolean to a number](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number#number_coercion): true is converted to 1, and false is converted to 0. Then compare the two operands loosely again.
4. Num and str =-> str covert to num
5. Big num n str => str convbert to num
6. const num = 0;

const big = 0n;

const str = "0";

const obj = new String("0");

console.log(num == str); // true

console.log(big == num); // true

console.log(str == big); // true

console.log(num == obj); // true

console.log(big == obj); // true

console.log(str == obj); // true

1. **Undefined**

* Declared but not assigned the value
* Variable declared but not initialized the js automatically assign the default undefined value
* typeof(undefined)

‘undefined’

1. **Null**

* Absence of value
* explicitly assigned by a programmer to indicate that a variable has no value
* typeof( null ) => object

1. **Closures**

* Function inside function is defined
* Inner function maintains reference to its outer function’s variable & can access them even after outer function has completed its execution
* E.g   
  function outer() {

const outerValue = 'Hi';

function inside() {

console.log('Accessing outer value inside the inner function ', outerValue);

}

return inside;  
}

const closureVal = outer();

closureVal();  
  
  
**Output** : Accessing outer value inside the inner function Hi

* outer function executes and returns the inside function and creates a closure where inside function maintains the reference to outside function
* Uses  
  1. Data privacy – lets we have counter variable so if we use it within closure the changes of its getting modified globally are reduced

2. Function factories – It is useful when you need multiple functions with similar logic but different configurations

1. **Var**

* Function or global scope
* Hoisted to the top with undefined value
* We can redeclare the var variable
* **E.g**var x = 1;

console.log(z)

// undefined

console.log(t)

// undefined

if (x === 1) {

var x = 2;

var x = 4;

var z;

z=0;

var t = 5;

console.log(x);

// Expected output: 2

}  
console.log(z);

// Expected output: 2

* **Output**   
  undefined  
  undefined  
  4  
  0

1. **Let**

* Block scope
* Cant be redeclare in same scope
* While hoisting not initialized any value so we can get reference error
* E.g  
  let str = 'snehal';

function getVal() {

console.log(str);

let str2 = 'mangale';

console.log(str2);

console.log(str3);

let str3 = 'suhas';

}

getVal()

**Output** :

snehal

mangale

Uncaught ReferenceError: Cannot access 'str3' before initialization

1. **Const**

* Can not reassigned the value
* Block scoped
* Has to initialized the value declaring
* Can not redeclare the value
* While hoisting not initialized any value so we can get reference error
* Hoisting happens within same block scope without initialising the value
* E.g

const name = 'snehal';

function getname() {

console.log(name);

const name2 = 'mangale';

console.log(name2);

console.log(name3);

const name3 = 'suhas';

}

getname()  
  
**Output** :

snehal

mangale

Uncaught ReferenceError: Cannot access 'name3' before initialization

1. **This keyword**

* This” keyword refers to an object that is executing the current piece of code.
* It references the object that is executing the current function.
* this – if nothing refers to window
* this with Object ‘
* **E.g**   
  let person = {

name:'snehal',

getName: function (city,zipcode) {

return this.name + city + zipcode;

}

}

const person3 = {

name:'mangale'

}

person.getName.call( person3,' Pune',' 411' );  
person.getName.apply( person3,[' Pune',' 411'] );  
  
Output   
// mangale Pune 411

* Call   
  - we pass owner object and then this will refer to that object  
  - list of arguments given as comma separated list
* Apply   
  - we pass owner object and then this will refer to that object  
  - list of arguments given as array
* Bind  
  - same as call and apply – we pass owner object to which this will be refer  
  - only difference is bind will return new function without executing it  
  - E.g   
  const personBind = person.getName.bind(person3, ' Pune',' 411')

personBind()  
  
// mangale Pune 411

1. **Difference between js functions and methods**

* Function  
   - Block of code written to perform any task  
  - We define function with keyword and pass some parameters if required  
  - we call the function to execute it
* Methods  
  - Property of object that contains function definition  
  - Object method can be accessed with following syntax  
  let person = {

name:'snehal',

getName: function (city,zipcode) {

return this.name + city + zipcode;

}

}

person.getName(' Pune',' 411')

* **Output** :

snehal Pune 411

1. **Synchronous and asynchronous**

<https://www.freecodecamp.org/news/synchronous-vs-asynchronous-in-javascript/>

1. **Array map function**

* Performs operations on each element and returns the new array
* It doesn’t make any modification in original array
* You can easily apply filter() with map().
* E.g  
  const array1 = [1,2,4,5,6]

array1.map((arr, index)=>arr\*2 )

* Output  
   [2, 4, 8, 10, 12]

1. **Array filter method**

* Execute function on each element of array
* It should return a [truthy](https://developer.mozilla.org/en-US/docs/Glossary/Truthy) value to keep the element in the resulting array, and a [falsy](https://developer.mozilla.org/en-US/docs/Glossary/Falsy) value otherwise.
* A [shallow copy](https://developer.mozilla.org/en-US/docs/Glossary/Shallow_copy) of the given array containing just the elements that pass the test. If no elements pass the test, an empty array is returned.
* Return array only with satisfied condition elements
* You can easily apply filter() with map().

1. **forEcah Method**

* forEach does not return anything
* We can use forEach when we are not going to any further logic on iterated values
* You can not apply filter() with forEach() because it returns undefined.
* E.g   
  let arr = [1,4,65,32,76]

arr.forEach((ele)=> {

if(ele > 40){

console.log(ele)

}

})  
Output   
65  
76

1. **Arrow functions**

* Arrow functions are better for short functions
* Arrow Functions Have an Implicit Return Statement
* No need of explicit return statement

**e.g   
const sum = (a,b)=> a+b**

* Arrow Functions Don’t Have this Binding
* In a regular function, the this keyword refers to the object from which you **call** the function. In an arrow function, the this keyword refers to the object from which you **define** the function.  
  e.g  
  const person = {

name: 'Nathan',

skills: ['HTML', 'CSS', 'JavaScript'],

showSkills() {

this.skills.forEach(skill => {

console.log(`${this.name} is skilled in ${skill}`);

});

},

};

person.showSkills();

**Output** :

Nathan is skilled in HTML

Nathan is skilled in CSS

Nathan is skilled in JavaScript

1. **Memoization**
2. **What is the difference between a JavaScript expression and a statement**

* Expression is unit of code that resolves to value
* Statement is unit of code that performs an action
* E.g
* // Statements

let x = 0;

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

if (true) { console.log('Hi'); }

// Expressions

x; // Resolves to 0

3 + x; // Resolves to 3

add(1, 2); // Resolves to 3

1. **Strict mode**

* Rules which are enforced in strict but not in regular mode
* **Undeclared variables**  
  - Regular mode won’t throw any error if we use undeclared variable  
  - E,.g   
   function getName(){  
   name= “snehal”  
   console.log( name )  
   }

Output  
 snehal

- Strict mode throws an error if we try to access undeclared variable  
- E.g  
 function getName(){  
 “use strict”

name = “snehal”

console.log( name )  
 }  
  
 getName()  
   
 Output   
 Refernceerror name not defined

* **Duplicate parameter name**  
  - In regular mode, Use of duplicate parameter name in function will take only second param value and first param value will be ignored  
  - E.g   
   function getSum(num,num){

return num + num;

}

getSum(3,5)  
  
 Output

10  
- In strict mode, throws an error saying Duplicate parameter name not allowed  
- E.g  
 function getSum(num,num){

"use strict"

return num + num;

}

getSum(3,5)

Output  
  
 Uncaught SyntaxError: Duplicate parameter name not allowed in this context

* **Use of reserved keywords**  
  - In regular mode, Use of reserved keyword will not throw any error  
  - In strict mode, use of reserve keyword will throw an error
* **Assignment to read only property**- In regular mode – if we try to assign new value to read only property then it will not throw an error but it will keep the old value and wont assign new value  
  - e.g   
   let obj = {};

Object.defineProperty(obj , 'key',{value:'sn',writtable:false});

obj.key = 'man'

obj  
  
 Output

{key: 'sn'}  
- In strictmode – if we try to assign value to read only property then it will throw an error  
 e.g  
  
"use strict"

let obj = {};

Object.defineProperty(obj , 'key',{value:'sn',writtable:false});

obj.key = 'man'

obj  
  
Output

VM779:4 Uncaught TypeError: Cannot assign to read only property 'key' of object '#<Object>'

1. **Prototypal inheritance**

* Prototype inheritance in javascript is the linking of prototypes of a parent object to a child object to share and utilize the properties of a parent class using a child class
* Prototypes are hidden objects that are used to share the properties and methods of a parent class with child classes.
* Each function is an object with property prototype which again an object
* Differential inheritance – Methods are not copied from parent to child. Instead, children have an invisible link back to their parent object
* E.g
* function Dog() {}Dog.prototype.bark = function() {
* console.log(“woof!”);
* };
* var fido = new Dog();
* fido.bark();
* // ‘woof!’
* For example, fido doesn’t actually have its own method called bark() (in other words, fido.hasOwnProperty(‘bark’) === false).
* What actually happens when I write fido.bark() is this:
* 1. The JS engine looks for a property called bark on our fido object.
* 2. It doesn’t find one, so it looks “up the prototype chain” to fido’s parent, which is Dog.prototype.
* 3. It finds Dog.prototype.bark, and calls it with this bound to fido.
* Object.create
* e.g
* var parent = {
* foo: function() {
* console.log(‘bar’);
* }
* };
* var child = Object.create( parent );
* child.hasOwnProperty(‘foo’); // false
* child.foo(); // ‘bar’
* Essentially, it creates a new, empty object that has parent in its prototype chain. That means that even though child doesn’t have its own foo() method, it has access to the foo() method from parent.
* When a function is used as a constructor, this refers to the new object that you’re creating
* E.g
* function Rectangle( width, height ) {
* this.width = width;
* this.height = height;
* }
* var rect = new Rectangle( 3, 4 );
* rect.width; // 3
* rect.height; // 4
* Rectangle.prototype.area = function(){
* return this.width \* this.height
* }
* var rect = new Rectangle( 3, 4 );
* rect.area(); // 12
* - There’s that this keyword again. Just like in the constructor, this inside of a method refers to the instance.

1. **Hoisting**

* The built in feature of JS through which functions, variables and classes declarations are moved to the top of their scope- all before code executions
* This allows us to use functions, variables and classes before they are declared
* Only declarations hoist to the top
* Initialization doesn’t hoist to top
* Javascript allocates memory before starting the execution
* E.g   
  console.log(z)

getName2()

var z=7

function getName2(){

console.log('snehal')

}

getName2()

getName2  
  
Output :

undefined

snehal

snehal

ƒ getName2(){

console.log('snehal')

}

* If we trying to execute **arrowfunction** before declaration it gives function not defined
* E.g

name()

var name = () => {

console.log('sn')

}  
  
Output :

Uncaught TypeError: name is not a function

1. **Promises**

* Promises are used to handle asynchronous operations
* Promises are used when you need to wait for task to finish before running next process
* We have resolve and reject function
* Either resolve or reject will get call
* The resolve function corresponds to then function, while reject corresponds to catch function
* E.g  
  let p = new Promise((resolve, reject) => {

let isTrue = true;

if(isTrue){

resolve('Promise resolved')

}

reject('Promise rejected')

})

p.then((res)=>{

console.log('Message rece ', res)

})

.catch((err)=>{

console.log('Error rece ', err)

})  
  
Output :

Message rece Promise resolved

* **Promise.all**  
  - Promise.all() method receives array of promises and return message after resolve   
  - All promises executed from the array as given sequence  
  E.g   
   const p1 = Promise.resolve('StartPromise One');  
   const p2 = Promise.resolve(200);  
   const p3 = Promise.resolve('Finished');

Promise.all([p1, p3, p2])

.then(messages => console.log(messages))

.catch(error => console.log(error));

Output :

['StartPromise One', 'Finished', 200]  
  
- If one of the promises is rejected, then the method returns the first rejection it encounters and stops any further process.  
- E.g   
const p1 = Promise.reject('Error From Promise One');

const p2 = Promise.resolve(200);

const p3 = Promise.resolve('Finished');

Promise.all([p1, p2, p3])

.then(messages => console.log(messages))

.catch(error => console.log(error));

Output:  
  
Error promise 2

* **Promise.allSettles()**  
  - Similar to all but when any promise rejected then method will store the rejected result and continue processing other promises  
  - When promises are settled, the method will return array of objects of details of each promise
* E.g  
    
  const p1 = Promise.resolve('Promise One');  
  const p2 = Promise.reject('Error promise 2');   
  const p3 = Promise.resolve('Finished');   
    
  Promise.allSettled([p1, p2, p3])   
  .then(messages => console.log(messages)) .  
  catch(error => console.log(error));
* Output :
* *(3) [{…}, {…}, {…}]*

**0**: {status: 'fulfilled', value: 'Promise One'}

**1**: {status: 'rejected', reason: 'Error promise 2'}

**2**: {status: 'fulfilled', value: 'Finished'}

* **Promise.any ()**  
  - The Promise.any() method is similar to the Promise.all() method, except that it returns **only a single value** from any promise that calls the **resolve() function first**  
  - E.g  
  const p1 = Promise.resolve('Promise One');

const p2 = Promise.reject('Error promise 2');

const p3 = Promise.resolve('Finished');

Promise.any([p1, p2, p3])

.then(messages => console.log(messages))

.catch(error => console.log(error));  
Output :

Promise One

* Promise.race()  
  -  the promise is settled when any promise is resolved or rejected:  
  - E.g   
  **Resolve e.g**  
    
  const p1 = Promise.resolve('Promise One');

const p2 = Promise.reject('Error promise 2');

const p3 = Promise.resolve('Finished');

Promise.race([p1, p2, p3])

.then(messages => console.log(messages))

.catch(error => console.log(error));  
  
Output :

Promise One  
  
**Reject E.g**  
const p1 = Promise.reject('Error Promise One');

const p2 = Promise.resolve(' promise 2');

const p3 = Promise.resolve('Finished');

Promise.race([p1, p2, p3])

.then(messages => console.log(messages))

.catch(error => console.log(error));  
  
Output :

Error Promise One

1. **Event delegation**

* E.g   
  <div>  
   <span>  
   <button>Click Me!</button>

</span>  
</div>

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

div.addEventListener("click", (event) => {

if(event.target.tagName === 'BUTTON') {

console.log("button was clicked")

}

})

* The event object has a target property which contains information about the element that actually received the event. On target.tagName, we get the name of the tag for the element, and we check if it's **BUTTON**.
* With this code, when you click the button, the event bubbles up to the div which handles the event.
* Due to event bubbling, when the button receives an event, say **click**, that event bubbles up the tree, so span and div will respectively receive the event also.
* The idea is that you "**delegate**" the handling of an event to a different element (in this case, the div, which is a parent element) instead of the actual element (the button) that received the event.
* With event delegation, you create fewer event listeners and perform similar events-based logic in one place.
* Event delegation is possible because of event propagation in the DOM, where the event a child element receives is also passed to the child's parent and ancestors

1. **Functional Programming**

* Coding style
* Programming paradigm
* Mindset – way of approaching the task
* **Why ?**
* Easier to debug or maintain as you are building a project
* **How?**
* Functions => taking input giving output
* Try to write code in functions
* **Avoid side effects => use pure functions** => means take inputs and computes value based on that inputs only  
  **pure function e.g**  
    
  function greet(name){  
   return “Hi I am, “ + name;  
  }
* Side effects means => function is doing something but not returning like consoling
* Or functions computes from global values and not only from inputes the function received
* **Higher order functions  
  -** functions can be input/output  
  - means function which can take other function as an input or return function as an output  
  - **E.g** function makeAdj(adjective){

return function(name){

return adjective + name;

}  
}

var adjective = makeAdj('cool ');

adjective('conference')  
  
Output :

cool conference

* **Don’t use iterate like for, while instead use map, reduce, filter**
* **Avoid mutability – use immutable data   
  -** Means cant be changed  
  - If we want to change anything in array we can use map on array which will create new array and wont change anything in original array  
  -It avoids lot of bugs because may be some other piece of code using the same array and if we are not touching original array it wont impact this piece of code
* **Persistent data structures for efficient immutability ( mori library, immutable.js library)**

1. **Destructuring**

* Feature introduced in ES6, which allows you to unpack the values from arrays or properties from objects
* Simplify your code, more readable and maintainable
* E. g array values read without destructuring  
  let arr = [1, 2]  
  let a = arr[0];  
  let b = arr[1];  
    
  E.g **with** **destructuring**let arr = [1, 2]  
  let [a, b] = arr;
* E.g Object properties reading without destructuring

Let person = {  
 name : “snehal”,  
 edu : “BE”  
}

let name = person.name;  
let edu = person.edu;  
  
E.g with destructuring  
  
let person = {  
 name: ‘snehal’,  
 edu : ‘BE’  
}  
  
let { name, edu } = person;

1. **Spread operator**

* **…** is the spread operator
* Allows iterable like array or string to be expanded into separate element , unpacks the element
* E.g **How spread operator works on array and string**

**let arr = [1, 2, 3, 4, 5];  
console.log(...arr)  
  
Output**

**1 2 3 4 5**

**let str = 'snehal'**

**console.log(...str)  
  
Output**

**s n e h a l**

* **E.g**  
   **without spread operator**   
    
  let arr = [1, 2, 3, 4, 5]

let maxNum = Math.max(arr)

console.log(maxNum)

**Output**   
  
NaN

**With apread operator**

let arr = [1, 2, 3, 4, 5]

let maxNum = Math.max(...arr) // spread operator unpack the elements

console.log(maxNum)  
  
**Output** :

5

* We can use spread operator to **make shallow copy** and then perform some operation on shallow copy so it **wont impact on original array**
* E.g   
    
  let fruits = ['apple', 'mango']

let fruits2 = [...fruits]  
fruits2[0]='apppppleeee'  
fruits  
Output :  
 ['apple', 'mango']

fruits2  
Output   
 ['apppppleeee', 'mango']

* **Merge two arrays**

E.g   
let fruits = ['apple', 'mango']

let vege = ['cabbage']

let fruitsAndVegies = [...fruits, ...vege]

fruitsAndVegies  
  
Output :

['apple', 'mango', 'cabbage']

1. **Static method**

* Methods are associated with class and not with instances
* E.g   
  class Person {

constructor(name,age){

this.name = name;

this.age = age;

}

getName(){

console.log('My name is ' + this.name);

}

static getAge(p1) {

console.log(p1.name)

}  
}

let p1 = new Person('Snehal', 29)

p1.getName()

Person.getAge(p1)

P1  
  
Output :

My name is Snehal

Snehal  
 *Person {name: 'Snehal', age: 29}*  
 **age**: 29  
 **name**: "Snehal"  
 [[Prototype]]: Object  
 **constructor**: *class Person*  
 **getName**: *ƒ getName()*

* As we can see above static methos is not part inx=stance p1

1. **Set object**

* lets you store **unique values** of any type, whether primitive values or object references.
* Set lets you **add, delete and checking the existence** of values in set
* E.g  
  const unique = new Set()

unique.add(5)

unique.add(10)

unique.add(5) // it will be ignored and wont be added as its duplicate

console.log(unique)  
Output;  
  
Set(2) {5, 10}  
**unique.size**

2

**unique.has(11)**

false

**unique.has(10)**

true

**unique.delete(10)**

true

**unique**

Set(1) {5}

* **Why we need ?**
* To remove duplicate values from array

E.g   
let arr = [1,4,5,2,1,4,5]  
let unique1 = new Set(arr)  
let uniqueArr = [...unique1]

Output :  
unique1

Set(4) {1, 4, 5, 2}  
  
uniqueArr

(4) [1, 4, 5, 2]

1. **Reduce method**

* Reduce the elements of an array to single value
* E.g

let prices = [20, 70, 80, 90]

let total = prices.reduce(sum)

console.log(‘Total ’,total)

function sum(accumulator, element){

console.log('acuumalator ' , accumulator , ' element ' , element)

return accumulator + element

}  
  
Output :

acuumalator 20 element 70

acuumalator 90 element 80

acuumalator 170 element 90

Total 260

* E.g   
  let prices2 = [20, 70, 80, 90]

let total2 = prices2.reduce((accu, ele)=>{

return accu + ele

})

Output :

total2

260

* It does not change the original array

1. **Sort method**

* let num = [10,6,30,7,8,6]
* num.sort((a,b)=> a-b)
* (6) [6, 6, 7, 8, 10, 30]
* let num = [10,6,30,7,8,6]
* num.sort((a,b)=> b-a)
* (6) [30, 10, 8, 7, 6, 6]
* let arrOfPerson = [{
* name:'snehal',
* age: 29
* },{
* name:'varsha',
* age: 28
* },{
* name:'anuradha',
* age: 32
* }]
* arrOfPerson.sort((a,b) => a.age - b.age)

{name: 'varsha', age: 28}

{name: 'snehal', age: 29}

{name: 'anuradha', age: 32}

arrOfPerson.sort((a,b) => b.age - a.age)

{name: 'anuradha', age: 32}

{name: 'snehal', age: 29}

{name: 'varsha', age: 28}

arrOfPerson.sort((a,b) => a.name.localeCompare( b.name))  
 {name: 'anuradha', age: 32}

{name: 'snehal', age: 29}

{name: 'varsha', age: 28}

arrOfPerson.sort((a,b) => b.name.localeCompare(a.name))

{name: 'varsha', age: 28}

{name: 'snehal', age: 29}

{name: 'anuradha', age: 32}