



SOFTGROWTH INFOTECH

JavaScript Interview Question



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 = 234567890123456789012345678901234567890n;
```

- **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;
  }
}
```

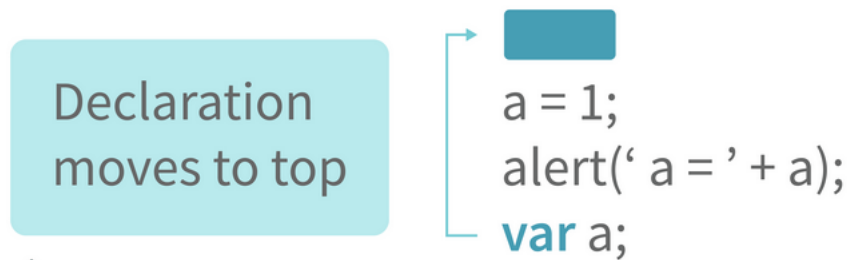
// Collection of data as an ordered list

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

3. 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. 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

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)
}
```

CH

- 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.
```

6. 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.

	var	let	const
origins	pre ES2015	ES2015(ES6)	ES2015(ES6)
scope	globally scoped OR function scoped. attached to window object	globally scoped OR block scoped	globally scoped OR block scoped
global scope	is attached to Window object.	not attached to Window object.	attached to Window object.
hoisting	var is hoisted to top of its execution (either global or function) and initialized as undefined	let is hoisted to top of its execution (either global or block) and left uninitialized	const is hoisted to top of its execution (either global or block) and left uninitialized
redeclaration within scope	yes	no	no
reassigned within scope	yes	yes	no

7. 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

```

8. 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.

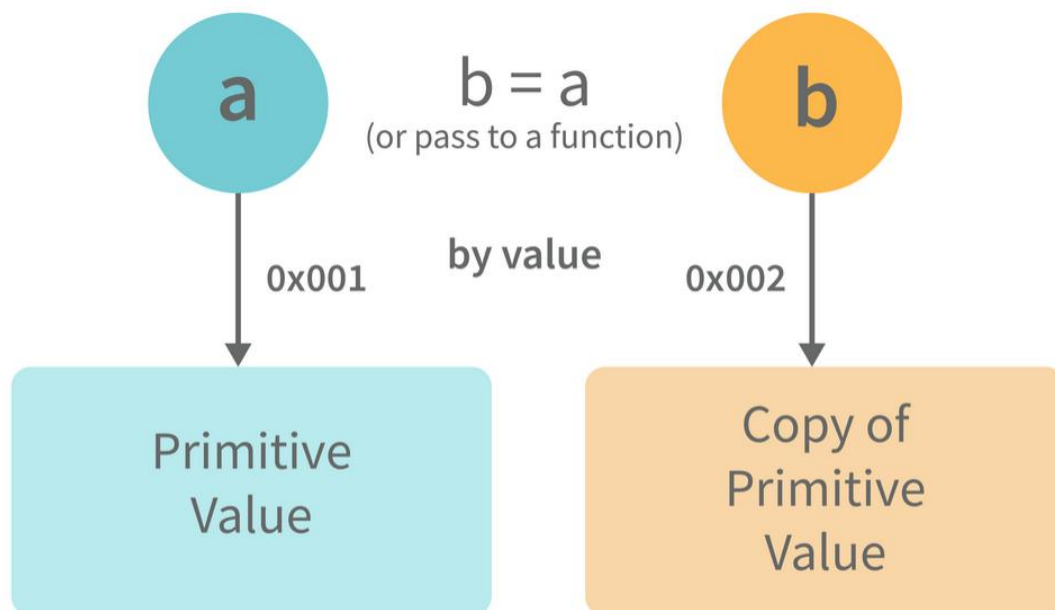
For understanding passed by value and passed by reference, we need to understand what happens when we create a variable and assign a value to it,

```
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
```

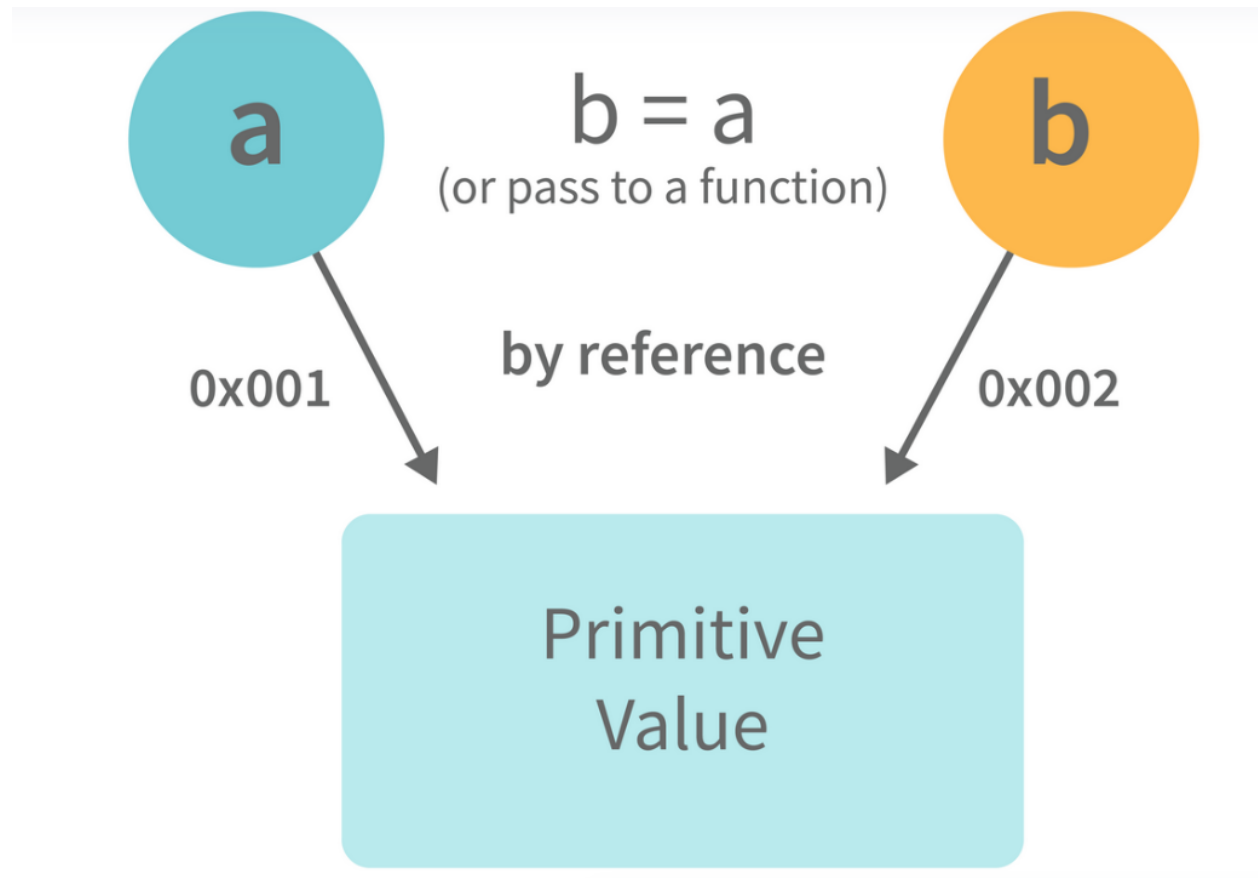
```
// Changing the value of y
```

```
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:



```
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**.

9. 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"
```

10. 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();
```

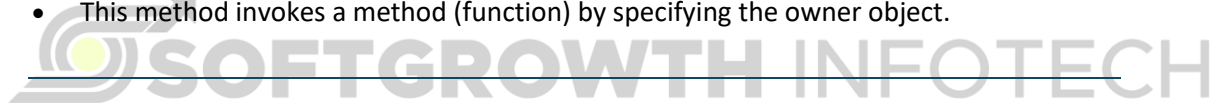
In the above code, at the time of invocation, the getName function is a property of the object **obj**, therefore, **this** keyword will refer to the object **obj**, and hence the output will be "vivek".

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

1.call

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 + " , " + brandName;
    }
}

var person1 = {name: "Vivek"};

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

// Binds the displayDetails function to the person1 object

detailsOfPerson1();

```

12. 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

13. 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

14. 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.

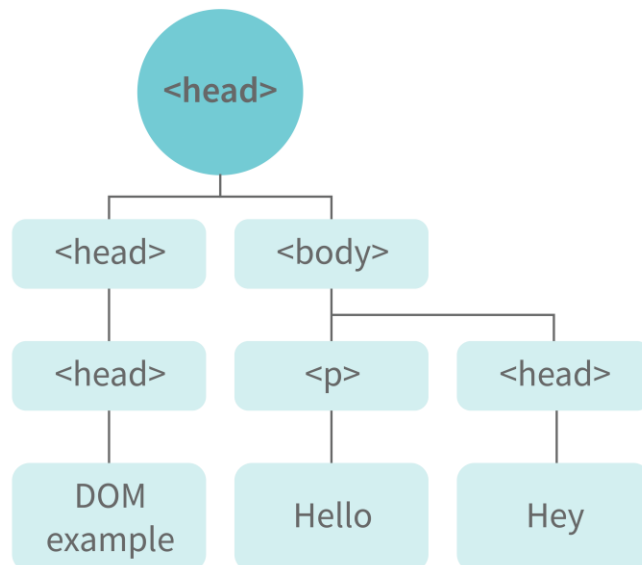
15. What is DOM?

- 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:*

```
<html>
<head>
<title>
DOM example
</title>
</head>
<body>
<p id = "para1">Hello</p>
<p id = "para2">Hey</p>
</body>
</html>
```



JavaScript Interview Questions for Experienced

16.What are arrow functions?

In ES6, arrow functions are introduced. The shorthand syntax for writing ES6 functions is arrow functions. The arrow function's definition consists of parameters, followed by an arrow (=>), and the function's body.

The 'fat arrow' function is another name for the Arrow function. We won't be able to employ them as constructors.

```
const function_name = (arg_1, arg_2, arg_3, ...) => {  
  //body of the function  
}
```

Few things to note:

- It reduces the size of the code.
- For a single-line function, the return statement is optional.
- Bind the context lexically.
- For a single-line statement, functional braces are not required.
- Doesn't work with **new**

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

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

18.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++;  
  }  
  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

19.What is hoisting in JavaScript?

Hoisting is a JavaScript concept that refers to the process of moving declarations to the top of their scope. This means that variables and functions can be used before they are declared, as long as they are declared before they are used in a function.

For example, the following code will print "Hello, world!" even though the greeting variable is not declared until after the console.log() statement.



```
var greeting = "Hello, world!";

function sayHello() {
  console.log(greeting);
}

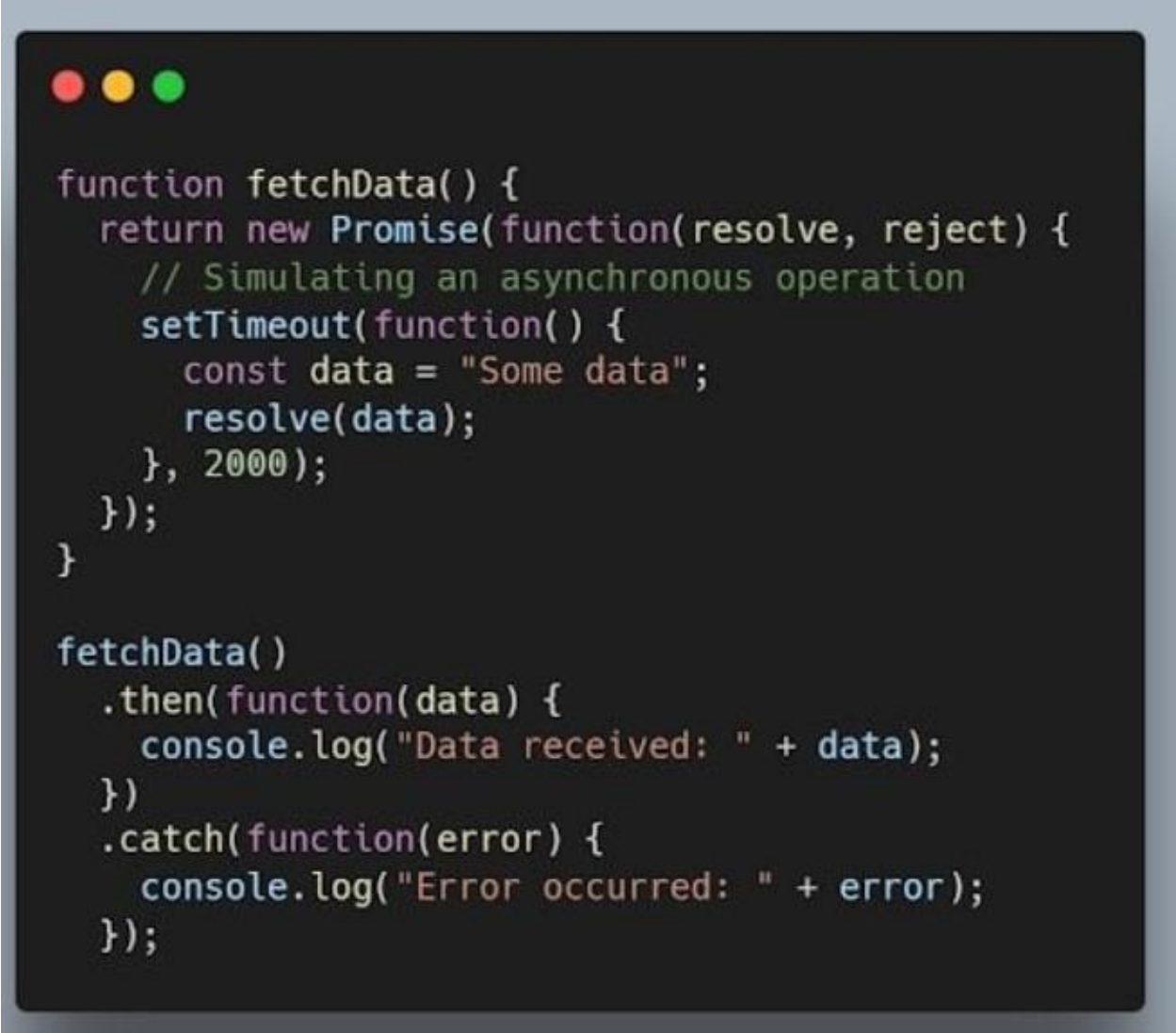
sayHello();
```

20.What is the difference between null and undefined?

21.What are promises in JavaScript?

Asynchronous programming is a concept found in JavaScript. The processes are run separately from the main thread in asynchronous programming. Promises are the most convenient approach to deal with asynchronous programming in ES6. Depending on the outcome of the procedure, a promise can be refused or resolved. Callbacks were used to cope with asynchronous programming before promises were introduced in ES6.

However, it caused the problem of callback hell, which was addressed with the introduction of promises. (A callback is a function that is performed after another function has completed. When working with events in JavaScript, callback is very useful. As an argument to another function, we pass a function into another function.)



```
function fetchData() {  
  return new Promise(function(resolve, reject) {  
    // Simulating an asynchronous operation  
    setTimeout(function() {  
      const data = "Some data";  
      resolve(data);  
    }, 2000);  
  });  
}  
  
fetchData()  
  .then(function(data) {  
    console.log("Data received: " + data);  
  })  
  .catch(function(error) {  
    console.log("Error occurred: " + error);  
  });
```

22.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

23. 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

24. 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

25.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

26.Give a thorough comparison between ES5 and ES6.

ES5	ES6
ES5 is the fifth edition of the ECMAScript which was introduced in 2009.	ES6 is the sixth edition of the ECMAScript which was introduced in 2015.
Primitive data types that are string , boolean , number , null , and undefined are supported by ES5.	There are a few additions to the JavaScript data types in ES6. For supporting unique values, a new primitive data type ' symbol ' was introduced.
In ES5, we could define the variables by using the var keyword only.	In ES6, in addition to var, there are two new methods to define variables: let and const .
Both function and return keywords are used in order to define a function in ES5.	An arrow function is a newly added feature in ES6 in which we don't require the function keyword in order to define the function.
In ES5, a for loop is used to iterate over elements.	ES6 introduced the idea of for...of loop in order to iterate over the values of the iterable objects.

27.What is the “spread” operator in ES6?

The list of parameters is obtained using the spread operator. Three dots (...) are used to represent it. The spread operator divides an iterable (such as an array or a string) into individual elements. It's mostly used in JavaScript to make shallow copies of JS. It improves the readability of your code by making it more concise.

The spread operator can be used to join two arrays together or to concatenate them.

```
let arr1 = [4, 5, 6];

let arr2 = [1, 2, 3, ...arr1, 7, 8, 9, 10];

console.log(arr2);
```

Output:

```
[ 1 2 3 4 5 6 7 8 9 10 ]
```

28. Discuss the template literals in ES6.

Template literals are a brand-new feature in ES6. It makes producing multiline strings and performing string interpolation simple.

Template literals, also known as string literals, allow for embedded expressions.

Template literals were referred to as template strings prior to ES6. The backtick (``) character is used to enclose template literals. The dollar sign and curly brackets (\${expression}) are used to denote placeholders in template literals. If we need to use an expression within the backticks, we can put it in the (\${expression}) variable.

```
let s1 = "Good";

let s2 = "Day";

let s = `${s1} ${s2}`;

console.log(s);
```

Output:

```
Good Day
```