JavaScript Interview Preparation Guide (50 Output related questions)

Frontend Developer Cheat sheet

*By **Ayush Verma** on 2021-04-13*

interview questions javascript

Contents

Question 1: (Strings, Numbers, Boolean)

```
var num = 8;
var num = 10;
console.log(num);
```

Answer 10 **Explanation — **With the var keyword, you can declare multiple variables with the same name. The variable will then hold the latest value. You cannot do this with let or const since they're block-scoped.

```
function sayHi() {
  console.log(name);
  console.log(age);
  var name = 'Ayush';
  let age = 21;
}

sayHi();
```

Answer undefined and ReferenceError **Explanation — **Within the function, we first declare the name variable with the var keyword. This means that the variable gets hoisted (memory space is set up during the creation phase) with the default value of undefined, until we actually get to the line where we define the variable. We haven't defined the variable yet on the line where we try to log the name variable, so it still holds the value of undefined.

Variables with the let keyword (and const) are hoisted, but unlike var, don't get *initialized*. They are not accessible before the line we declare (initialize) them. This is called the "temporal dead zone". When we try to access the variables before they are declared, JavaScript throws a ReferenceError.

Question 3:

```
function getAge() {
  'use strict';
  age = 21;
  console.log(age);
}
```

Answer ReferenceError **Explanation** With "use strict", you can make sure that you don't accidentally declare global variables. We never declared the variable age, and since we use "use strict", it will throw a reference error. If we didn't use "use strict", it would have worked, since the property age would have gotten added to the global object.

Question 4:

```
+true;
!'Ayush';
```

Answer 1 and false **Explanation** The unary plus tries to convert an operand to a number. true is 1, and false is 0.

Question 5:

```
let number = 0;
console.log(number++);
console.log(++number);
console.log(number);
```

Answer 0 2 2. **Explanation** The postfix unary operator ++:

- Returns the value (this returns 0).

- Increments the value (number is now 1).

- Increments the value (number is now 2).

- Returns the value (this returns 2).

This returns 0 2 2.

Question 6:

```
function sum(a, b) {
  return a + b;
}

sum(1, '2');
```

Answer "12" **Explanation** JavaScript is a dynamically typed language: we don’t specify what types of certain variables are. Values can automatically be converted into another type without you knowing, which is called *implicit type coercion*. Coercion is converting from one type into another.

In this example, JavaScript converts the number 1 into a string, in order for the function to make sense and return a value. During the addition of a numeric type (1) and a string type ('2'), the number is treated as a string. We can concatenate strings like "Hello" + "World", so what's happening here is "1" + "2" which returns "12".

Question 7:

```
const name = 'Ayush';

name.giveAyushPizza();
```

Answer "Just give Ayush pizza already!" **Explanation** String is a built-in constructor, which we can add properties to. I just added a method to its prototype. Primitive strings are automatically converted into a string object, generated by the string prototype function. So, all strings (string objects) have access to that method!

Question 8:

```
for (let i = 1; i < 5; i++) {
  if (i === 3) continue;
  console.log(i);
}
```

Answer 1 2 4 **Explanation** The continue statement skips an iteration if a certain condition returns true.

Question 9:

```
function sayHi() {
  return (() => 0)();
}

console.log(typeof sayHi());
```

Answer "number" **Explanation** The sayHi function returns the returned value of the immediately invoked function expression (IIFE). This function returned 0, which is type "number".

FYI: there are only 7 built-in types: null, undefined, boolean, number, string, object, and symbol. "function" is not a

Question 10:

```
console.log(typeof typeof 1);
```

Answer "string" **Explanation** typeof 1 returns "number". And typeof "number" returns "string".

Question 11:

```
!!null;
!!'';
!!1;
```

Answer false false true **Explanation** null is falsy. !null returns true. !true returns false.

"" is falsy. !"" returns true. !true returns false.

1 is truthy. !1 returns false. !false returns true.

Question 12:

```
[...'Ayush'];
```

Answer ["A", "y", "u", "s", "h"] **Explanation** A string is an iterable. The spread operator maps every character of an iterable to one element.

Question 13:

```
console.log(3 + 4 + '5');
```

Answer "75" **Explanation — **Operator associativity is the order in which the compiler evaluates the expressions, either left-to-right or right-to-left. This only happens if all operators have the *same* precedence. We only have one type of operator: +. For addition, the associativity is left-to-right.

3 + 4 gets evaluated first. This results in the number 7.

7 + '5' results in "75" because of coercion. JavaScript converts the number 7 into a string. We can concatenate two strings using the +operator. "7" + "5" results in "75".

Question 14:

```
var a = 10;
var b = a;
b = 20;
```

```
var a = 'Ayush';
var b = a;
b = 'Verma';

console.log(a);
console.log(b);
```

Answer — **1. 10 and 20 2. "Ayush" and "Verma" **Explanation The value assigned to the variable of primitive data type is **tightly coupled**. That means, whenever you create a copy of a variable of primitive data type, the value is copied to a new memory location to which the new variable is pointing to. When you make a copy, it will be a **real copy.**

Question 15:

```
function sum(){
  return arguments.reduce((a, b) => a + b);
}

console.log(sum(1,2,3)); (1)

function sum(...arguments){
  return arguments.reduce((a, b) => a + b);
```

Answer — **1. Error will be thrown. 2. 6 **Explanation —

- Arguments are not fully functional array, they have only one method length. Other methods cannot be used on them.
- ... rest operator creates an array of all functions parameters. We then use this to return the sum of them.

Question 16:

```
console.log(1 == '1');
console.log(false == '0');
console.log(true == '1');
console.log('1' == '01');
console.log(10 == 5 + 5);
```

Answer true true true false true. **Explanation** —'1' == '01' as we are comparing two strings here they are different but all other equal.

Question 17:

```
console.log('1' - - '1'); (1)
console.log('1' + - '1'); (2)
```

Answer — **1. 2 2. “1-1” **Explanation

- With type coercion string is converted to number and are treated as 1 - -1 = 2. 2.+ operator is used for concatenation of strings in javascript, so it is evaluated as '1' + '-1' = 1-1.

Question 18:

```
let lang = 'javascript';
(function(){
   let lang = 'java';
})();

console.log(lang); (1)

(function(){
   var lang2 = 'java';
})();

console.log(lang2); (2)
```

Answer — **1. “javascript” 2. Error will be thrown. **Explanation —

- Variables defined with let are blocked scope and are not added to the global object.
- Variables declared with var keyword are function scoped, so wrapping the function inside a closure will restrict it from being accessed outside that is why it throws error

Question 19:

```
(function(){
   console.log(typeof this);
}).call(10);
```

Answer object** Explanation** — call invokes the function with new this which in this case is 10 which is basically a constructor of Number and Number is object in javascript.

Question 20:

```
console.log("[ayushv.medium.com/](https://ayushv.medium.com/)" instanceof String); (1)

const s = new String('[ayushv.medium.com/](https://ayushv.medium.com/)');
console.log(s instanceof String); (2)
```

Answer — **1. false 2. true **Explanation — Only strings defined with String() constructor are instance of it.

Question 21: (Objects, Arrays)

```
const obj = { a: 'one', b: 'two', a: 'three' };
console.log(obj);
```

Answer { a: "three", b: "two"}" **Explanation** If you have two keys with the same name, the key will be replaced. It

Question 22:

```
let c = { greeting: 'Hey!' };
let d;

d = c;
c.greeting = 'Hello';
console.log(d.greeting);
```

Answer Hello **Explanation — **In JavaScript, all objects interact by *reference* when setting them equal to each other.

First, a variable c holds a value to an object. Later, we assign d with the same reference that c has to the object. When you change one object, you change all of them.

Question 23:

```
let a = 3;
let b = new Number(3);
let c = 3;

console.log(a == b);
```

Answer true false false **Explanation** new Number() is a built-in function constructor. Although it looks like a number, it's not really a number: it has a bunch of extra features and is an object.

When we use the == operator, it only checks whether it has the same *value*. They both have the value of 3, so it returns true.

However, when we use the === operator, both value *and* type should be the same. It's not: new Number() is not a number, it's an object. Both return false.

Question 24:

```
function getAge(...args) {
  console.log(typeof args);
}

getAge(21);
```

Answer "object" **Explanation** The rest parameter (...args) lets us "collect" all remaining arguments into an array. An array is an object, so typeof args returns "object".

Question 25:

```
console.log(greetign);
```

Answer {} **Explanation** It logs the object, because we just created an empty object on the global object! When we mistyped greeting as greetign, the JS interpreter actually saw this as global.greetign = {} (or window.greetign = {} in a browser).

In order to avoid this, we can use "use strict". This makes sure that you have declared a variable before setting it equal to anything.

Question 26:

```
function checkAge(data) {
  if (data === { age: 18 }) {
    console.log('You are an adult!');
  } else if (data == { age: 18 }) {
    console.log('You are still an adult.');
  } else {
    console.log(`Hmm.. You don't have an age I guess`);
  }
}

checkAge({ age: 18 });
```

the same location in memory.

The two objects that we are comparing don’t have that: the object we passed as a parameter refers to a different location in memory than the object we used in order to check equality.

This is why both { age: 18 } === { age: 18 } and { age: 18 } == { age: 18 } return false.

Question 27:

```
const obj = { 1: 'a', 2: 'b', 3: 'c' };
const set = new Set([1, 2, 3, 4, 5]);

obj.hasOwnProperty('1');
obj.hasOwnProperty(1);
set.has('1');
set.has(1);
```

Answer true true false true **Explanation** All object keys (excluding Symbols) are strings under the hood, even if you don’t type it yourself as a string. This is why obj.hasOwnProperty('1') also returns true.

It doesn’t work that way for a set. There is no '1' in our set: set.has('1') returns false. It has the numeric type 1, set.has(1) returns true.

```
const a = {};
const b = { key: 'b' };
const c = { key: 'c' };

a[b] = 123;
a[c] = 456;

console.log(a[b]);
```

Answer 456 **Explanation** Object keys are automatically converted into strings. We are trying to set an object as a key to object a, with the value of 123.

However, when we stringify an object, it becomes "[object Object]". So what we are saying here, is that a["[object Object]"] = 123. Then, we can try to do the same again. c is another object that we are implicitly stringifying. So then, a["[object Object]"] = 456.

Then, we log a[b], which is actually a["[object Object]"]. We just set that to 456, so it returns 456.

Question 29:

```
const numbers = [1, 2, 3];
numbers[10] = 11;
console.log(numbers);
```

Answer [1, 2, 3, 7 x empty, 11] **Explanation** When you set a value to an element in an array that exceeds the length of the array, JavaScript creates something called “empty slots”. These actually have the value of undefined, but you will see something like:

[1, 2, 3, 7 x empty, 11]

depending on where you run it (it’s different for every browser, node, etc.).

Question 30:

```
let person = { name: 'Ayush' };
const members = [person];
person = null;

console.log(members);
```

Answer [{ name: "Ayush" }] **Explanation** We are only modifying the value of the person variable, and not the first element in the array, since that element has a different (copied) reference to the object. The first element in members still holds its reference to the original object. When we log the members array, the first element still holds the value of the object, which gets logged.

```
const person = {
  name: 'Ayush',
  age: 21,
};

for (const item in person) {
  console.log(item);
}
```

Answer "name", "age" **Explanation** With a for-in loop, we can iterate through object keys, in this case name and age. Under the hood, object keys are strings (if they're not a Symbol). On every loop, we set the value of item equal to the current key it's iterating over. First, item is equal to name, and gets logged. Then, item is equal to age, which gets logged.

Question 32:

```
[1, 2, 3].map(num => {
  if (typeof num === 'number') return;
  return num * 2;
});
```

Answer [undefined, undefined, undefined] **Explanation** When mapping over the array, the value of num is equal statement typeof num === "number" returns true. The map function creates a new array and inserts the values returned from the function.

However, we don’t return a value. When we don’t return a value from the function, the function returns undefined. For every element in the array, the function block gets called, so for each element, we return undefined.

Question 33:

```
var obj = {a:1};
var secondObj = obj;
secondObj.a = 2;

console.log(obj);
console.log(secondObj);

var obj = {a:1};
var secondObj = obj;
secondObj = {a:2};

console.log(obj);
console.log(secondObj);
```

Answer — **1. { a:2 }and { a:2 } 2. { a:1 }and { a:2 } Explanation — **1. If the object property is changed, then reference) 2. If the object is reassigned with a new object then it is allocated to a new memory location, i.e it will be a **real copy** (call by value).

Question 34:

```
const arrTest = [10, 20, 30, 40, 50][1, 3];
console.log(arrTest);
```

Answer 40** Explanation** The last element from the second array is used as the index to get the value from first array like arrTest[3].

Question 35:

```
console.log([] + []);                (1)
console.log([1] + []);               (2)
console.log([1] + "abc");            (3)
console.log([1, 2, 3] + [1, 3, 4]); (4)
```

**Answer — **1. "" 2. "1" 3. "1abc" 2. "1,2,31,3,4" **Explanation — **1. An empty array is while printing in console.log is treated as Array.toString(), so it prints an empty string. 2. An empty array when printed in console.log is treated as Array.toString() and so it is basically "1" + "" = "". 3. "1" + "abc" = "1abc". 4 "1, 2, 3" + "1

Question 36:

```
const ans1 = NaN === NaN;
const ans2 = Object.is(NaN, NaN);
console.log(ans1, ans2);
```

Answer false true **Explanation** NaN is a unique value so it fails in equality check, but it is the same object so Object.is returns true.

Question 37:

```
var a = 3;
var b = {
  a: 9,
  b: ++a
};
console.log(a + b.a + ++b.b);
```

Answer 18 **Explanation — **Prefix operator increments the number and then returns it. So the following expression will be evaluated as 4 + 9 + 5 = 18.

```
const arr = [1, 2, undefined, NaN, null, false, true, "", 'abc', 3];
console.log(arr.filter(Boolean)); (1)

const arr = [1, 2, undefined, NaN, null, false, true, "", 'abc', 3];
console.log(arr.filter(!Boolean)); (2)
```

**Answer —

- **[1, 2, true, “abc”, 3].**
- **It will throw an error. **Explanation — **1. Array.filter() returns the array which matches the condition. As we have passed Boolean it returned all the truthy value.
- As Array.filter() accepts a function, !Boolean returns false which is not a function so it throws an error Uncaught TypeError: false is not a function.

Question 39:

```
const person = {
  name: 'Ayush Verma',
  .25e2: 25
};

console.log(person[25]);
console.log(person[.25e2])
```

Answer 25 25 undefined** Explanation** While assign the key the object evaluates the numerical expression so it becomes person[.25e2] = person[25]. Thus while accessing when we use 25 and .25e2 it returns the value but for '.25e2' is undefined.

Question 40:

```
console.log(new Array(3).toString());
```

Answer “,,”** Explanation** Array.toString() creates string of the array with comma separated values.

Question 41: (setTimeout and “this” keyword)

```
const foo = () => console.log('First');
const bar = () => setTimeout(() => console.log('Second'));
const baz = () => console.log('Third');

bar();
foo();
baz();
```

Answer First Third Second **Explanation** We have a setTimeout function and invoked it first. Yet, it was logged

This is because, in browsers, we don't just have the runtime engine, we also have something called a WebAPI. The WebAPI gives us the setTimeout function to start with and for example the DOM. The WebAPI can't just add stuff to the stack whenever it's ready. Instead, it pushes the callback function to something called the *queue*. An event loop looks at the stack and task queue. If the stack is empty, it takes the first thing on the queue and pushes it onto the stack.

Question 42:

```
for (var i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 1);
}

for (let i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 1);
}
```

Answer 3 3 3 and 0 1 2 **Explanation — **Because of the event queue in JavaScript, the setTimeout callback function is called *after* the loop has been executed. Since the variable i in the first loop was declared using the var keyword, this value was global. During the loop, we incremented the value of i by 1 each time, using the unary operator ++. By the time the setTimeout callback function was invoked, i was equal to 3 in the first example.

each value is scoped inside the loop.

Question 43:

```
let obj = {
    x: 2,
    getX: function() {
        setTimeout(() => console.log('a'), 0);
        new Promise( res =>  res(1)).then(v => console.log(v));
        setTimeout(() => console.log('b'), 0);
    }
}

obj.getX();
```

Answer 1 a b** Explanation** When a macrotask is completed, all other microtasks are executed in turn first, and then the next macrotask is executed.

Mircotasks include: MutationObserver, Promise.then() and Promise.catch(), other techniques based on Promise such as the fetch API, V8 garbage collection process, process.nextTick() in node environment.

Marcotasks include initial script, setTimeout, setInterval, setImmediate, I/O, UI rendering.

An immediately resolved promise is processed faster than an immediate timer because of the event loop the task queue (which stores timed out setTimeout() callbacks).

Question 44:

```
const shape = {
  radius: 10,
  diameter() {
    return this.radius * 2;
  },
  perimeter: () => 2 * Math.PI * this.radius,
};

console.log(shape.diameter());
console.log(shape.perimeter());
```

Answer 20 and NaN **Explanation** Note that the value of diameter is a regular function, whereas the value of perimeter is an arrow function.

With arrow functions, the this keyword refers to its current surrounding scope, unlike regular functions! This means that when we call perimeter, it doesn't refer to the shape object, but to its surrounding scope (window for example).

There is no value radius on that object, which returns NaN.

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

const member = new Person('Ayush', 'Verma');
Person.getFullName = function() {
  return `${this.firstName} ${this.lastName}`;
};

console.log(member.getFullName());
```

Answer TypeError **Explanation** In JavaScript, functions are objects, and therefore, the method getFullName gets added to the constructor function object itself. For that reason, we can call Person.getFullName(), but member.getFullName throws a TypeError.

If you want a method to be available to all object instances, you have to add it to the prototype property:

```
Person.prototype.getFullName = function() {
  return `${this.firstName} ${this.lastName}`;
};
```

Question 46:

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

const ayush = new Person('Ayush', 'Verma');
const sarah = Person('Sarah', 'Smith');

console.log(ayush);
console.log(sarah);
```

Answer Person {firstName: "Ayush", lastName: "Verma"} and undefined **Explanation** For sarah, we didn't use the new keyword. When using new, this refers to the new empty object we create. However, if you don't add new, this refers to the global object!

global.firstName = "Sarah" and global.lastName = "Smith". sarah itself is left undefined, since we don't return a value from the Person function.

Question 47:

```
const person = { name: 'Ayush' };
```

```
}

console.log(sayHi.call(person, 21));
console.log(sayHi.bind(person, 21));
```

Answer Ayush is 21 function **Explanation** With both, we can pass the object to which we want the this keyword to refer. However, .call is also *executed immediately*!

.bind. returns a *copy* of the function, but with a bound context! It is not executed immediately.

Question 48:

```
let obj = {
    x: 2,
    getX: function() {
        console.log(this.x);
    }
}

obj.getX(); (1)

let x = 5;
let obj = {
    x: 2,
    getX:() => {
```

```
obj.getX(); (2)

let x = 5;
let obj = {
    x: 2,
    getX: function(){
        let x = 10;
        console.log(this.x);
    }
}

let y = obj.getX;
y(); (3)
```

Answer — **1) 2 2) 5 3) 5 **Explanation First case is a regular function, the this keyword is bound to different values based on the ***context*** in which the function is called. Here obj is calling the function to this will point to current obj.

The second case is an arrow function, it will use the value of this in their ***lexical scope **i.e *value of x in surrounding scope. Here surrounding is the global scope or window object and "x" is also present. If "x" is not present then it is undefined.

In the third case, "y" is assigned a value of obj.getX,and "y" is in the global scope or window object. Hence "this"

Question 49:

```
let a = 10, b = 20;
setTimeout(function () {
  console.log('Ayush');
  a++;
  b++;
  console.log(a + b);
});
console.log(a + b);
```

Answer 30 “Ayush” 32. **Explanation** Settimeout pushes the function into BOM stack in event loop or it is executed after everything is executed in main function. So the results are printed after the console.log of main function.

Question 50:

```
function a() {
  this.site = 'Ayush';

  function b(){
    console.log(this.site);
```

```
var site = 'Wikipedia';
a(); (1)

function a() {
  this.site = 'Ayush';

  function b(){
    console.log(this.site);
  }

  b();
}

var site = 'Wikipedia';
new a(); (2)

function a() {
  this.site = 'Ayush';

  function b(){
    console.log(this.site);
  }

  b();
}
```

**Answer —

- 'Ayush'.
- 'Wikipedia'
- undefined

Explanation

- When a function with normal syntax is executed the value of this of the nearest parent will be used if not set at execution time, so in this case it is window, we are then updating setting the property site in the window object and accessing it inside so it is 'Ayush'.
- When a function is invoked as a constructor with new keyword then the value of this will be the new object which will be created at execution time so currently, it is { site: 'Ayush' }.

But when a function with normal syntax is executed, the value of this will default to global scope if it is not assigned at the execution time so it is window for b() and var site = 'Wikipedia'; adds the value to the global object, hence when it is accessed inside b() it prints 'Wikipedia'.

- Variables defined with let are not added to the global scope. Hence, undefined.

I hope you have found this useful. Thank you for reading!

CONTINUE LEARNING

How to Set Timeout with the JavaScript Fetch API using AbortController

abortcontroller fetch api settimeout javascript

Add Custom Markers with the Google Maps JavaScript API

An updated solution on how to add Google Maps to your website using JavaScript and tweaking it with customer markers

google maps javascript

How to Convert a Date String to Timestamp in JavaScript?

Different ways of converting a date string to a timestamp in JavaScript.

timestamp javascript

Analysing the Big O of various Array and Object methods

arrays big o objects performance javascript

How to Create and Style a div Using JavaScript?

css div javascript

How to Find the Odd Numbers in an Array with JavaScript

coding javascript programming web development array

Sign up for our free weekly newsletter

Your email address*

Subscribe

Powered by Formula