TOP 50 JAVASCRIPT INTERVIEW QUESTIONS

With Example Answers

GRAPHiC<>DAR

TOP 50 JAVASCRIPT INTERVIEW QUESTIONS WITH EXAMPLE ANSWERS

Review these common JavaScript interview questions and answers and practice your coding fundamentals with this guide for your next interview.

Preparing for a JavaScript interview requires a lot of work. It's important to be well-known in the fundamentals but you also should have some grasp on how to debug JavaScript code, what some of the advanced functions are and how to build projects in it.

COMMON JAVASCRIPT INTERVIEW QUESTIONS

1. What are the different data types in JavaScript?
2. What is hoisting in JavaScript?
3. What is the difference between null and undefined?
4. What are closures in JavaScript?
5. What is a callback function in JavaScript?
6. What are promises in JavaScript?
7. What is the purpose of the setTimeout() function in Javascript?
8. How can you check if an array includes a certain value?
9. How can you remove duplicates in an array?
10. What is the purpose of async and await in JavaScript?

JAVASCRIPT FUNDAMENTALS

1. What is JavaScript?

A high-level, interpreted programming language called JavaScript makes it possible to create interactive web pages and online apps with dynamic functionality. Commonly referred to as the universal language, Javascript is primarily used by developers for front-end and back-end work.

---

2. What are the different data types in JavaScript?

Primitive data types can store only a single value. To store multiple and complex values, **non-primitive data types** are used.

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

JavaScript has six primitive data types:

- Number
- String
- Boolean
- Null
- Undefined
- Symbol

It also has two compound data types:

- Object
- Array

3. 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 function is declared after calling function.

```
hoistedFunction();

// Outputs " Hello world! " even  when the function is declared after calling function

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

4. What is the difference between null and undefined?

null is an assignment value that represents no value or an empty value, while **undefined** is a variable that has been declared but not assigned a value.

```
var x;
console.log(x);   // Output : undefined

var y = null;
console.log(y);  // Output : null
```

5. Why do we use the word “debugger” in JavaScript?

The word “**debugger**” is used in JavaScript to refer to a tool that can be used to step through JavaScript code line by line. This can be helpful for debugging JavaScript code, which is the process of finding and fixing errors in JavaScript code.

To use the **debugger**, you need to open the JavaScript console in your browser. Then, you can use **debugger** commands to comb through your code line by line.

It's essential to know debugging techniques as well as the more general ideas behind code optimization and speed improvement. In addition to operating smoothly, efficient code significantly enhances the user experience.

For example, the following code will print the value of the x variable at each step of the **debugger**.

```
var x = 10;
debugger;
x = x + 1;
debugger;
console.log(x);
```

6. What is the purpose of the “this” keyword in JavaScript?

The **this** keyword refers to the object that is executing the current function or method.

It allows access to object properties and methods within the context of that object.

```
const person ={
  name : “Jonh”,
  greet : funtion(){
    console.log(“Hello, “ + this.name);
  }
}

person.greet();  // Output : Hello, Jonh
```

7. What is the difference between == and === operators in JavaScript?

The equality **==** operator is a comparison operator that compares two values and returns true if they are equal. The strict equality **===** operator is also a comparison operator, but it compares two values and returns true only if they are equal and of the same type.

For example, the following code will return true, because the values of the x and y variables are equal.

```
var x = 10;
var y = 10;
console.log(x == y);  // Output : true
console.log(x === y); // Output : false
```

8. What is the difference between “var” and “let” keywords in JavaScript?

The **var** and **let** keywords are both used to declare variables in JavaScript. However, there are some key differences between the two keywords.

The **var** keyword declares a global variable, which means that the variable can be accessed from anywhere in the code. The **let** keyword declares a local variable, which means that the variable can only be accessed within the block of code where it is declared.

```
{
  let x = 10;
  console.log(x);  // Output : 10
}
```

9. What are closures in JavaScript?

- Closures (**closureFn**) are functions that have access to variables from an outer function even after the outer function has finished executing. They "remember" the environment in which they were created.

```
function outer(){
 var outerVar = "Hello";
 function inner(){
   console.log(outerVar):
 }
 return inner;
}
var clouserFn = outer();
clouserFn();   //Output : Hello
```

10. What is event delegation in JavaScript?

- **Event delegation** is a technique where you attach a single event listener to a parent element, and that event listener handles events occurring on its child elements. It helps optimize performance and reduce memory consumption.

11. What is the difference between “let”, “const”, and “var”?

let and **const** were introduced in ES6 and have block scope. **let** is reassignable, and **const** is non-reassignable. **var** is function-scoped and can be redeclared and reassigned throughout the function.

```
let x = 5;
x = 10;
console.log(x);  //Output : 10

const y  = 5;
y = 10;     //Error: Assignment to constant variable
console.log(y);

var x = 5;
var x = 10;
console.log(x);   //Output : 10
```

12. What is implicit type coercion in JavaScript?

Implicit type coercion is a JavaScript concept that refers to the automatic conversion of a value from one type to another. In JavaScript, this conversion follows a priority order that typically begins with strings, then numbers, and finally Booleans. If you try to add a string to a number, JavaScript will implicitly coerce the number to a string before performing the addition operation because strings have the highest priority in type coercion.

For example, when you combine the number 5 with the string '10' using the addition operator, the result is the string '510'. This occurs because JavaScript will implicitly convert the number 5 to a string following the priority of coercion, and then concatenate it to the string '10'.

```
var x = 5;
var y = "10";
console.log(x + y);   //Output : "510"
```

13. Explain the concept of prototypes in JavaScript.

Prototypes are a mechanism used by JavaScript objects for inheritance. Every JavaScript object has a prototype, which provides properties and methods that can be accessed by that object.

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

Person.prototype.greet : function(){
 console.log("Hello, " + this.name);
}

var person = new Person("Jonh");
person.greet();  //Output: Hello, Jonh
```

14. What is the output of the following code?

```
console.log(3 + 2 + “7”);
```

The output will be "**57**". The addition operation is performed from left to right, and when a string is encountered, it performs concatenation.

---

15. How can you clone an object in JavaScript?

There are multiple ways to clone an object in JavaScript. One common method is using the **Object.assign()** method or the **spread operator (...).**

```
const obj1 = {name: “Jonh”, age: 30};

// Using Object.assign()
const obj2 = Object.assign({},obj1);

// Using Spread Operator
const obj3 = {...obj1};

console.log(obj2);     //Output: { name: “Jonh”, age: 30}

console.log(obj3);     //Output: { name: “Jonh”, age:  30}
```

INTERMEDIATE CONCEPTS

16. What are higher-order functions in JavaScript?

Higher order functions are functions that can accept other functions as arguments or return functions as their results. They enable powerful functional programming patterns in JavaScript.

```
function multiplybyTwo(num){
  num * 2;
}
function applyOperation(num, operation){
 return operation(num);
}
const result = applyOperation(5, multiplybyTwo);
console.log(result);    //Output: 10
```

17. What is the purpose of the bind() method in JavaScript?

The **bind()** method is used to create a new function with a specified **this** value and an initial set of arguments. It allows you to set the context of a function permanently.

```
const person ={
  name : "Jonh",
  greet : funtion(){
    console.log("Hello, " + this.name);
  }
}

const greetFn = person.greet;
const  boundFn = greetFn.bind(person);
boundFn();    // Output : Hello, Jonh
```

18. What is the difference between function declarations and function expressions?

- **Function declarations** are defined using the function keyword, while **function expressions** are defined by assigning a function to a variable. Function declarations are hoisted, while function expressions are not.

```
// Function Declaration
function multiply(a, b){
  return a * b;
}
// Function Expression
const add = function(a, b){
 return a + b;
}
```

19. What are the different types of errors in JavaScript?

JavaScript can throw a variety of errors, including:

- **Syntax errors:** These errors occur when the JavaScript code is not syntactically correct.
- **Runtime errors:** These errors occur when the JavaScript code is executed and there is a problem.
- **Logical errors:** These errors occur when the JavaScript code does not do what it is supposed to do.

---

20. What is memoization in JavaScript?

Memoization is a technique that can be used to improve the performance of JavaScript code.

Memoization works by storing the results of expensive calculations in a cache. This allows the JavaScript code to avoid re-performing the expensive calculations if the same input is provided again.

For example, the following code calculates the factorial of a number. The factorial of a number is the product of all the positive integers from one to the number.

```
function factorial(n){
  if(n === 0){
    return 1;
  }else{
    return n * factorial(n - 1);
  }
}
```

This code can be memoized as follows:

```
function factorial(n){
  if(factorialCache[n] !== undefined){
    return factorialCache[n];
  }else{
    factorialCache[n] = n * factorial(n - 1);
    return factorialCache[n];
  }
}
```

21. What is recursion in JavaScript?

Recursion is a programming technique that allows a function to call itself. Recursion can be used to solve a variety of problems, such as finding the factorial of a number or calculating the Fibonacci sequence.

The following code shows how to use recursion to calculate the factorial of a number:

```
function factorial(n){
  if(n === 0){
    return 1;
  }else{
    return n * factorial(n - 1);
  }
}
```

22. What is the use of a constructor function in JavaScript?

A **constructor function** is a special type of function that is used to create objects. Constructor functions are used to define the properties and methods of an object.

The following code shows how to create a constructor function:

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

23. What is the difference between a function declaration and a function expression in JavaScript?

A **function declaration** is a statement that defines a function. A **function expression** is an expression that evaluates to a function.

The following code shows an example of a function declaration. This code defines a function named factorial. The factorial function calculates the factorial of a number.

```
function factorial(n){
  if(n === 0){
    return 1;
  }else{
    return n * factorial(n - 1);
  }
}
```

The following code shows an example of a function expression:

```
var factorial = function(n){
  if(n === 0){
    return 1;
  }else{
    return n * factorial(n - 1);
  }
}
```

---

24. What is a callback function in JavaScript?

A **callback function** is a function passed as an argument to another function, which is then invoked inside the outer function. It allows asynchronous or event-driven programming.

```
function fetchData(callback){
 setTimeOut( functio(){
   const data = “Some Data”;
   callback(data);
 }, 2000);
}
function processData(data){
  console.log(“Data Received: ” + data);
}
fetchData(processData);  //Output: after 2
seconds: Data Received: Some Data
```

25. What are promises in JavaScript?

Promises are objects used for asynchronous operations. They represent the eventual completion or failure of an asynchronous operation and allow chaining and handling of success or error cases.

```
function fetchData(){
  return new Promise(function(resolve, reject ){
    setTimeOut( functio(){
      const data = “Some Data”;
      callback(data);
    }, 2000);
  }
}
fetchData()
.then(function(data){
  console.log(“Data Received: ” + data);
})
.catch(function(error)){
  console.log(“Error Occured: ” + error);
});
)
```

26. What is the difference between synchronous and asynchronous programming?

In **synchronous programming**, the program execution occurs sequentially, and each statement blocks the execution until it is completed. In **asynchronous programming**, multiple tasks can be executed concurrently, and the program doesn't wait for a task to finish before moving to the next one.

Synchronous coding example:

```
console.log("Statement 1");
console.log("Statement 2");
console.log("Statement 3");
```

Asynchronous code example:

```
console.log("Statement 1");
 setTimeOut( functio(){
   console.log("Statement 2");
}, 2000);
console.log("Statement 3");
```

---

27. How do you handle errors in JavaScript?

Errors in JavaScript can be handled using **try-catch** blocks. The try block contains the code that may throw an error, and the catch block handles the error and provides an alternative execution path.