

Top 30 JavaScript Interview Questions and Answers for 2024



Ravi Sharma · Follow

29 min read · Jan 1, 2024



Listen



Share



Top 30 JavaScript Interview Questions and Answers for 2024

Prepare for your next JavaScript interview with confidence!

Whether you're a seasoned developer or just starting your career in tech, this essential resource for 2024 will help you brush up on core concepts, from basic language features to advanced topics.

In this article, I've compiled over 30 of the most crucial JavaScript interview questions along with detailed answers and code examples.

Dive into this invaluable collection to ace your JavaScript interviews and stay ahead in today's competitive tech industry”

• • •

💡 Nail JavaScript interviews with the right practice questions and in-depth solutions from ex-interviewers! [Try GreatFrontEnd](#) → 💡

• • •

Open in app ↗

Sign up

Sign in

Medium



Search



3. Explain the event loop in JavaScript and how it helps in asynchronous programming.

4. Difference between var, let, and const ?

5. Different data types in Javascript?

6. What is callback function and callback hell ?

7. What is Promise and Promise chaining?

8. What is async/await ?

9. What is the difference between == and === operators ?

10. Different ways to create an Object in Javascript ?

11. What is rest and spread operator?

12. What is a higher-order function?

Level-2 : Intermediate

13. What is Closure? What are the use cases of Closures?

14. Explain the concept of hoisting in JavaScript.

15. What is a Temporal dead zone?
16. What is a prototype chain? and Object.create() method?
17. What is the difference between Call, Apply, and Bind methods?
18. What are lambda or arrow functions?
19. What is the currying function?
20. What are the features of ES6?

Level-3: Expert

21. What is Execution context, execution stack, variable object, scope chain?
22. What is the priority of execution of callback, promise, setTimeout, process.nextTick()?
23. What is the Factory function and generator function?
24. Different ways to clone (Shallow and deep copy of object) an object?
25. How to make an object immutable? (seal and freeze methods)?
26. What is Event and event flow, event bubbling and event capturing?
27. What is Event delegation?
28. What are server-sent events?
29. What is a web worker or service worker in javascript?
30. How to compare 2 JSON objects in javascript?

=====

1. Is Javascript single-threaded?

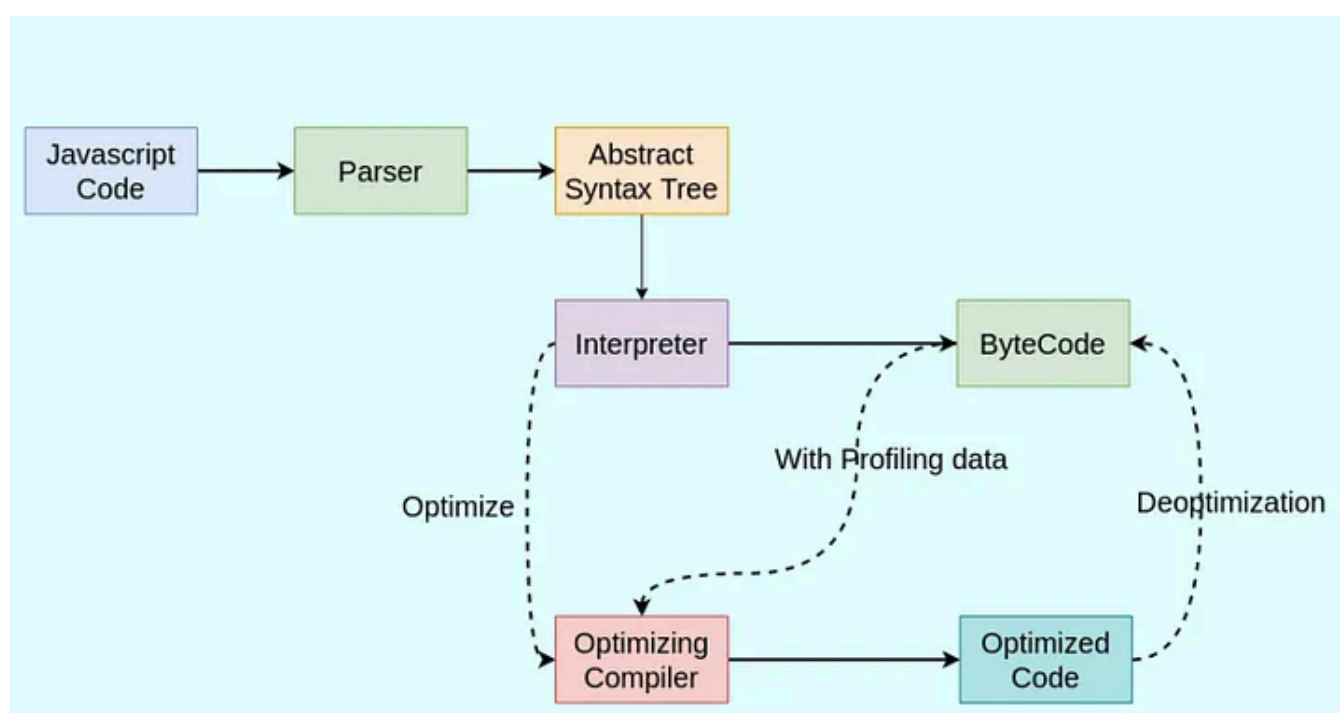
Yes, JavaScript is a **single-threaded** language. This means that it has **only one call stack** and **one memory heap**. Only one set of instructions is executed at a time.

Also, Javascript is **Synchronous and blocking** in nature. meaning that code is executed line by line and one task must be completed before the next one begins

However, JavaScript also has asynchronous capabilities, which allow certain operations to be executed independently of the main execution thread. This is commonly achieved through mechanisms like callbacks, promises, async/await, and event listeners. These asynchronous features enable JavaScript to handle tasks such as fetching data, handling user input, and performing I/O operations without blocking the main thread, making it suitable for building responsive and interactive web applications.

2. Explain the main component of the JavaScript Engine and how it works.

Every browser has a Javascript engine that executes the javascript code and converts it into machine code.



When JavaScript code is executed, the parser first reads the code and produces an AST, and stores it in memory. The interpreter then processes this AST and generates bytecode or machine code, which is executed by the computer.

The profiler is a component of a JavaScript engine that **monitors** the execution of the code.

Bytecode is used by optimizing the compiler along with profiling data. "Optimizing compiler" or Just-in-time (JIT) compiler makes certain assumptions based on profiling data and produces highly optimized machine code.

Sometimes there is a case where the 'optimization' assumption is incorrect and then it goes back to the previous version via the "Deoptimize" phase (where it actually

becomes the overhead to us)

JS Engine usually optimizes “hot functions” and uses inline caching techniques to optimize the code.

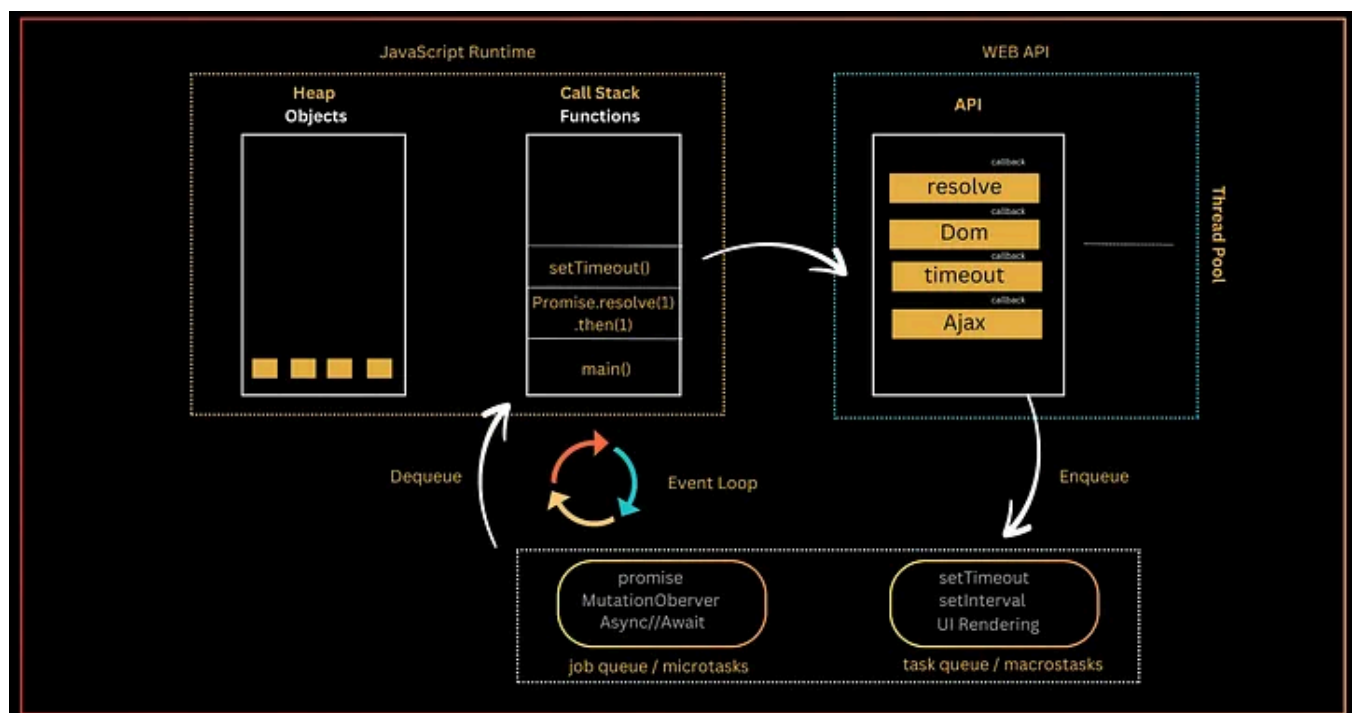
During this process, the call stack keeps track of the currently executing functions, and the memory heap is used for memory allocation.

Finally, the garbage collector comes into play to manage memory by reclaiming memory from unused objects.

Google Chrome V8 Engine:

1. Interpreter is called “Ignition”.
2. Optimizing compiler is called “TurboFan”.
3. Apart from Parser, there is a “pre-parser” that checks for syntax and tokens
4. “Sparkplug” is introduced which is present between “Ignition” & “TurboFan” which is also called Fast Compiler.

3. Explain the Event loop in JavaScript.



The Event loop is a core component of the JavaScript runtime environment. It is responsible for scheduling and executing asynchronous tasks. The event loop works by continuously monitoring two queues: the call stack and the event queue.

The call stack is a stack(LIFO) data structure that stores the functions that are currently being executed (stores the execution context created during the code execution).

Web APIs is the place where the async operations (setTimeout, fetch requests, promises) with their callbacks are waiting to complete. **It borrows the thread from the thread pool to complete the task in the background** without blocking the main thread.

The job queue (or microtasks) is a FIFO (First In, First Out) structure that **holds the callbacks of async/await, promises, process.nextTick()** that are ready to be executed. For example, the resolve or reject callbacks of a fulfilled promise are enqueued in the job queue.

The task queue (or macrotasks) is a FIFO (First In, First Out) structure that **holds the callbacks of async operations (timer like setInterval, setTimeout)** that are ready to be executed. For example, the callback of a timed-out `setTimeout()` — ready to be executed — is enqueued in the task queue.

The Event loop permanently monitors whether the call stack is empty. If the call stack is empty, the event loop looks into the job queue or task queue and **dequeues any callback ready to be executed into the call stack.**

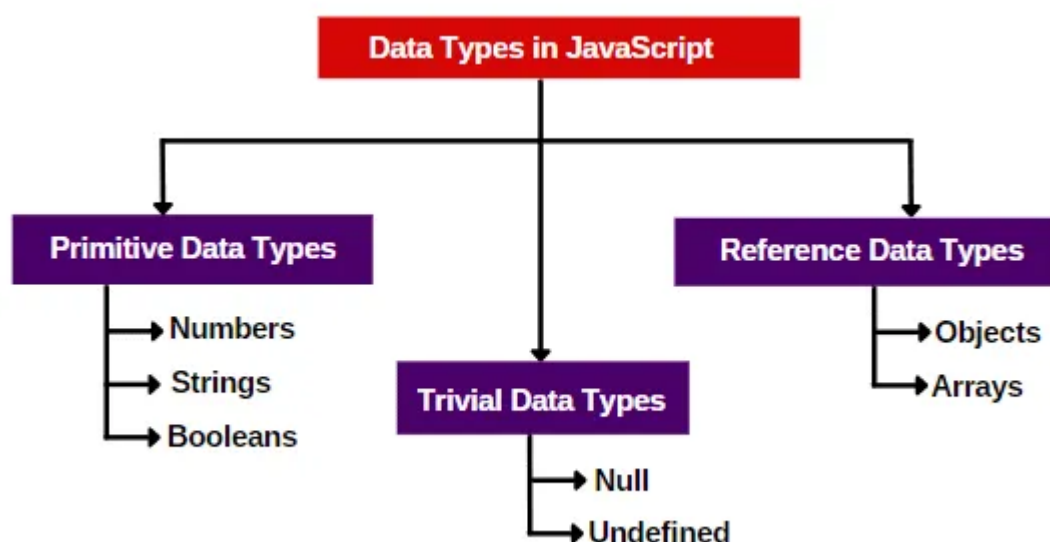
4. Difference between var, let, and const?

	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 <i>undefined</i>	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

In a browser the **window object** is the window of the browser, the top structure in the HTML tree. Variables declared with **var** globally are attached to the **window object**. Type `var dog = 'browser'` in the browser's console and then type `window.dog`. The value 'browser' appears! This makes controlling the scope of the variable even more difficult. By contrast, **let** and **const** are not attached to the window object.

5. Different data types in Javascript?

JavaScript is a dynamic and loosely typed, or duck-typed language. It means that we do not need to specify the type of variable because the JavaScript engine dynamically determines the data type of a variable based on its values.



Primitive data types in JavaScript are the most basic data types that **represent single values**. They are **immutable (cannot be changed)** and directly hold a specific value.

In JavaScript, a **Symbol** is a **primitive data type** introduced in ECMAScript 6 (ES6) that **represents a unique and immutable value**. It is often used as an **identifier for object properties** to avoid name collisions

```
const mySymbol = Symbol('key');
const obj = {
  [mySymbol]: 'value'
};
```

When a Symbol is used as a property key, it doesn't clash with other property keys, including string keys.

6. What is callback function and callback hell ?

In JavaScript, callbacks are commonly used to handle asynchronous operations.

Callback function is a function that is **passed as an argument** to another function and is intended to be **executed after the completion of a specific task or at a given time**.

```
function fetchData(url, callback) {
  // Simulate fetching data from a server
  setTimeout(() => {
    const data = 'Some data from the server';
    callback(data);
  }, 1000);
}

function processData(data) {
  console.log('Processing data:', data);
}

fetchData('https://example.com/data', processData);
```

In this example, the `fetchData` function takes a URL and a **callback function as arguments**. After fetching the data from the server (simulated using `setTimeout`), it calls the callback function and passes the retrieved data to it.

Callback Hell, also known as “**Pyramid of Doom**” is a term used in JavaScript programming to describe a situation where multiple nested callbacks are used within asynchronous functions.

“It occurs when asynchronous operations depend on the results of previous asynchronous operations, resulting in deeply nested and often hard-to-read code.”

Callback Hell is an anti-pattern with multiple nested callbacks which makes code hard to read and debug when dealing with asynchronous logic.

```
fs.readFile('file1.txt', 'utf8', function (err, data) {  
  if (err) {  
    console.error(err);  
  } else {  
    fs.readFile('file2.txt', 'utf8', function (err, data) {  
      if (err) {  
        console.error(err);  
      } else {  
        fs.readFile('file3.txt', 'utf8', function (err, data) {  
          if (err) {  
            console.error(err);  
          } else {  
            // Continue with more nested callbacks...  
          }  
        });  
      }  
    });  
  }  
});  
});
```

In this example, we’re reading three files sequentially using the `fs.readFile` function, and each file reading operation is asynchronous. As a result, we have to nest the callbacks inside one another, creating a pyramid structure of callbacks.

To avoid Callback Hell, modern JavaScript provides alternatives like Promises and `async/await`. Here’s the same code using Promises:

```
const readFile = (file) => {  
  return new Promise((resolve, reject) => {  
    fs.readFile(file, 'utf8', (err, data) => {  
      if (err) {
```

```
        reject(err);
      } else {
        resolve(data);
      }
    });
  });
};
```

```
readFile('file1.txt')
  .then((data1) => {
    return readFile('file2.txt');
  })
  .then((data2) => {
    return readFile('file3.txt');
  })
  .then((data3) => {
    // Continue with more promise-based code...
  })
  .catch((err) => {
    console.error(err);
  });
```

7. What is Promise and Promise chaining?

Promise: A Promise is an object in JavaScript used for asynchronous computations. It represents the result of an asynchronous operation, the result may be resolved or rejected.

Promises have three states:

1. **Pending:** The initial state. This is the state in which the Promise's eventual value is not yet available.
2. **Fulfilled:** The state in which the Promise has been resolved successfully, and the eventual value is now available.
3. **Rejected:** The state in which the Promise has encountered an error or has been rejected, and the eventual value cannot be provided.

Promise constructor has two parameters (**resolve, reject**) which are functions. If the async task has been completed without errors then call the resolve function with message or fetched data to resolve the promise.

If an error occurred then call the reject function and pass the error to it.

we can access the result of promise using `.then()` handler.

we can catch the error in the `.catch()` handler.

```
// Creating a Promise
const fetchData = new Promise((resolve, reject) => {
  // Simulate fetching data from a server
  setTimeout(() => {
    const data = 'Some data from the server';
    // Resolve the Promise with the retrieved data
    resolve(data);
    // Reject the Promise with an error
    // reject(new Error('Failed to fetch data'));
  }, 1000);
});

// Consuming the Promise
fetchData
  .then((data) => {
    console.log('Data fetched:', data);
  })
  .catch((error) => {
    console.error('Error fetching data:', error);
  });
```

Promise chaining: The process of executing a sequence of asynchronous tasks one after another using promises is known as Promise chaining.

It involves chaining multiple `.then()` methods to a Promise to perform a series of tasks in a specific order.

```
new Promise(function (resolve, reject) {
  setTimeout(() => resolve(1), 1000);
})
  .then(function (result) {
    console.log(result); // 1
    return result * 2;
  })
  .then(function (result) {
    console.log(result); // 2
    return result * 3;
  })
  .then(function (result) {
    console.log(result); // 6
```

```
    return result * 4;
  });
```

8. What is async/await ?

Async/await is a **modern approach** to handling asynchronous code in JavaScript. It provides a more concise and readable way to work with Promises and async operations, effectively avoiding the “Callback Hell” and improving the overall structure of asynchronous code.

In JavaScript, the **async** keyword is used to define an asynchronous function, which returns a Promise.

Within an async function, the **await** keyword is used to pause the execution of the function until a Promise is resolved, effectively allowing for synchronous-looking code while working with asynchronous operations.

```
async function fetchData() {
  try {
    const data = await fetch('https://example.com/data');
    const jsonData = await data.json();
    return jsonData;
  } catch (error) {
    throw error;
  }
}

// Using the async function
fetchData()
  .then((jsonData) => {
    // Handle the retrieved data
  })
  .catch((error) => {
    // Handle errors
  });
```

In this example, the `fetchData` function is defined as an async function, and it uses the `await` keyword to pause the execution and wait for the `fetch` and `json` operations, effectively working with Promises in a way that resembles synchronous code.

9. What is the difference between == and === operators ?

== (Loose Equality Operator): This operator performs type coercion, which means it converts the operands to the same type before making the comparison. It checks if the values are equal without considering their data types. For example, `1 == '1'` will return `true` because JavaScript converts the string `'1'` to a number before comparison.

=== (Strict Equality Operator): This operator performs a strict comparison without type coercion. It checks if both the values and their data types are equal. For example, `1 === '1'` will return `false` because the data types are different (number and string).

In summary, `==` checks for equality after type coercion, whereas `===` checks for strict equality, considering both the values and their data types.

Execution of `==` will be fast as compared to the `===` statement.

Some of the example that covers the above cases:

```
0 == false // true
0 === false // false
1 == "1" // true
1 === "1" // false
null == undefined // true
null === undefined // false
'0' == false // true
'0' === false // false
[] == [] or [] === [] //false, refer different objects in memory
{} == {} or {} === {} //false, refer different objects in memory
```

10. Different ways to create an Object in Javascript ?

In JavaScript, there are several ways to create objects. Some common methods for object creation include:

a)Object Literals: The most straightforward way to create an object is by using object literals, which define an object's properties and methods in a comma-separated list enclosed in curly braces.

```
let person = {
  firstName: 'John',
```

```
    lastName: 'Doe',  
    greet: function() {  
        return 'Hello, ' + this.firstName + ' ' + this.lastName;  
    }  
};
```

b)Constructor Function: Constructor functions can be used to create multiple instances of an object with the new keyword. Inside the constructor function, properties and methods can be assigned to the this keyword.

```
function Person(firstName, lastName) {  
    this.firstName = firstName;  
    this.lastName = lastName;  
    this.greet = function() {  
        return 'Hello, ' + this.firstName + ' ' + this.lastName;  
    };  
}  
  
let person1 = new Person('John', 'Doe');  
let person2 = new Person('Jane', 'Smith');
```

c)Object.create(): The Object.create() method allows you to create a new object with a specified prototype object. This method provides more control over the prototype of the newly created object.

```
let personProto = {  
    greet: function() {  
        return 'Hello, ' + this.firstName + ' ' + this.lastName;  
    }  
};  
  
let person = Object.create(personProto);  
person.firstName = 'John';  
person.lastName = 'Doe';
```

d)Class Syntax (ES6): With the introduction of ES6, JavaScript supports class syntax for defining objects using the class keyword. This provides a more familiar and structured way to create objects and define their properties and methods.

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
  greet() {
    return 'Hello, ' + this.firstName + ' ' + this.lastName;
  }
}

let person = new Person('John', 'Doe');
```

e)Factory Functions: Factory functions are functions that return an object. This approach allows you to encapsulate the object creation process and easily create multiple instances with custom properties.

```
function createPerson(firstName, lastName) {
  return {
    firstName: firstName,
    lastName: lastName,
    greet: function() {
      return 'Hello, ' + this.firstName + ' ' + this.lastName;
    }
  };
}

let person1 = createPerson('John', 'Doe');
let person2 = createPerson('Jane', 'Smith');
```

f)Object.setPrototypeOf(): The Object.setPrototypeOf() method can be used to set the prototype of a specified object. This offers an alternative approach to setting the prototype of an object after its creation.

```
let personProto = {
  greet: function() {
    return 'Hello, ' + this.firstName + ' ' + this.lastName;
  }
};

let person = {};
person.firstName = 'John';
```

```
person.lastName = 'Doe';  
Object.setPrototypeOf(person, personProto);
```

g)Object.assign(): The Object.assign() method can be used to create a new object by copying the values of all enumerable own properties from one or more source objects to a target object. This is particularly useful for merging objects or creating a shallow copy.

```
let target = { a: 1, b: 2 };  
let source = { b: 3, c: 4 };  
let mergedObject = Object.assign({}, target, source);
```

h)Prototype Inheritance: JavaScript uses prototypal inheritance, allowing objects to inherit properties and methods from other objects. You can create objects by leveraging prototypal inheritance and using the prototype property of constructor functions or classes to define shared behavior.

```
function Animal(name) {  
    this.name = name;  
}  
  
Animal.prototype.greet = function() {  
    return 'Hello, I am ' + this.name;  
};  
  
function Dog(name, breed) {  
    Animal.call(this, name);  
    this.breed = breed;  
}  
  
Dog.prototype = Object.create(Animal.prototype);  
Dog.prototype.constructor = Dog;  
  
let myDog = new Dog('Max', 'Poodle');
```

i)Singleton Pattern: The singleton pattern is used to restrict an object to a single instance. It can be implemented in JavaScript using a combination of closures and

immediately invoked function expressions (IIFE). This ensures that only one instance of the object is created.

```
let singleton = (() => {
  let instance;

  function createInstance() {
    return {
      // properties and methods
    };
  }

  return {
    getInstance: () => {
      if (!instance) {
        instance = createInstance();
      }
      return instance;
    }
  };
})();
```

11. What is rest and spread operator?

The rest operator, represented by three dots (`...`), is used in function parameters to **collect a variable number of arguments into an array**. It allows you to pass an arbitrary number of arguments to a function **without explicitly defining them as named parameters**.

```
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}
console.log(sum(1, 2, 3, 4)); // Outputs 10
```

The spread operator, also denoted by three dots (`...`), is used to **spread the elements of an array or object into another array or object**. It allows you to easily clone arrays, concatenate arrays, and merge objects.

```
const array1 = [1, 2, 3];
const array2 = [4, 5, 6];
```

```
const mergedArray = [...array1, ...array2];  
// mergedArray is [1, 2, 3, 4, 5, 6]
```

```
const obj1 = { a: 1, b: 2 };  
const obj2 = { b: 3, c: 4 };  
const mergedObject = { ...obj1, ...obj2 };  
// mergedObject is { a: 1, b: 3, c: 4 }
```

12. What is a higher-order function?

Higher-order function in JavaScript is a function that either takes one or more functions as arguments or returns a function as its result. In other words, it operates on functions, either by taking them as arguments, returning them, or both.

```
function operationOnArray(arr, operation) {  
  let result = [];  
  for (let element of arr) {  
    result.push(operation(element));  
  }  
  return result;  
}  
  
function double(x) {  
  return x * 2;  
}  
  
let numbers = [1, 2, 3, 4];  
let doubledNumbers = operationOnArray(numbers, double);  
console.log(doubledNumbers); // Output: [2, 4, 6, 8]
```

They enable powerful techniques such as function composition, currying, and callback-based asynchronous operations. Understanding higher-order functions is essential for writing expressive and functional-style JavaScript code.

An unary function (i.e. monadic) is a function that accepts exactly one argument. It stands for a single argument accepted by a function.

13. What is Closure? What are the use cases of Closures?

Closure is a feature that allows the function to capture the environment (or to retain access to variables from the scope) where it is defined, even after that scope has closed.

We can say the closure is a **combination of a function and lexical environment where that function is defined.**

In other words, a closure gives a function access to its own scope, the scope of its outer function, and the global scope, allowing it to “remember” and continue to access variables and parameters from these scopes.

```
function outerFunction() {  
  let outerVariable = 'I am from the outer function';  
  
  return innerFunction() {  
    console.log(outerVariable); // Accessing outerVariable from the outer funct  
  }  
}  
  
let myFunction = outerFunction();  
myFunction(); // Output: I am from the outer function
```

Closure is created every time when a function is created at the time of function creation and when you define a function inside another function.

Execution context is an environment where JavaScript code is executed. For each function call a separate execution context is created and pushed into the execution stack. Once function execution completed it is popped off from stack.

Every execution context has a space in memory where its variables and function are stored, and once the function popped off from the execution stack a JavaScript Garbage collector clear all of these things.

In JavaScript, anything is garbage-collected only when there are no references to it.

In the above example, the anonymous execution context still has a reference to the variables to the memory space of its outer environment. Even though the `outerFunction()` is finished. (It can access the `outerVariable` variable and use it inside `console.log(outerVariable)`).

Closures have several important use cases in JavaScript:

1. **Data Privacy and Encapsulation:** Closures can be used to create private data and encapsulate functionality within a limited scope. By defining functions within another function, the inner functions have access to the outer function's variables, but these variables are inaccessible from outside the outer function. This allows for the creation of private data and methods that are not directly accessible from the outside, thereby enhancing data privacy and encapsulation.
2. **Maintaining State:** Closures are often used to maintain state in asynchronous operations and event handling. For example, when handling asynchronous tasks, closures can capture and retain the state of variables across multiple asynchronous operations, ensuring that the correct variables are accessed when the asynchronous tasks complete.
3. **Currying and Partial Application:** Closures facilitate functional programming techniques such as currying and partial application. By using closures to capture and remember specific parameters and return a new function that uses these captured parameters, currying and partial application can be achieved. This allows for the creation of specialized functions with pre-set arguments, providing flexibility and reusability.
4. **Module Pattern:** Closures are essential in implementing the module pattern in JavaScript. By using closures to create private variables and expose only the necessary public methods, developers can create modular and organized code, preventing unwanted access and modification of internal module data.
5. **Callback Functions:** Closures are often employed when working with callback functions. A closure can be used to capture and maintain the state of variables within the context of an asynchronous operation, ensuring that the correct variables are accessible when the callback function is invoked.

14. Explain the concept of hoisting in JavaScript.

Hoisting in JavaScript is the **default behavior** where variable and function **declarations are moved to the top of their containing scope** during the **compilation phase, before the actual code execution**. This means that you can use a variable or call a function before it's declared in your code.

When you declare a variable using `var`, the declaration is hoisted to the top of its containing function or block and **initialized with the default value of "undefined"**.

```
console.log(x); // Outputs: undefined
var x = 5;
```

Variables declared with `let` and `const` are hoisted as well but have a "temporal dead zone" where they cannot be accessed before their declaration.

```
console.log(x); // Throws an error (ReferenceError)
let x = 5;
```

Function declarations are also hoisted to the top of their containing scope. You can call a function before it's declared in your code.

```
sayHello(); // Outputs: "Hello, world!"
function sayHello() {
  console.log("Hello, world!");
}
```

Hoisting is not happening with an arrow function, function expression, or variable initialization.

15. What is a Temporal dead zone?

The Temporal Dead Zone (TDZ) is a concept in JavaScript related to variable declarations using `let` and `const`.

When you declare a variable with `let` or `const`, it is hoisted to the top of its containing scope. However, unlike `var`, variables declared with `let` and `const` remain uninitialized in the TDZ.

Any attempt to access or use the variable before its actual declaration within the scope will result in a `ReferenceError`. This is to prevent the use of variables before they have been properly defined.

Understanding the Temporal Dead Zone is important because it helps prevent bugs related to variable usage before initialization. It also promotes best practices in

JavaScript coding by encouraging proper variable declarations before use.

16. What is a prototype chain? and Object.create() method?

In JavaScript, every function and object has a property named `prototype` by default.

Every object in JavaScript has a prototype. A prototype is another object from which the current object inherits properties and methods. You can think of the prototype as a template or a parent object.

The prototype chain is a mechanism that allows objects to inherit properties and methods from other objects

When you access a property or method on an object, JavaScript first looks for it on the object itself. If it doesn't find it, it looks up the prototype chain until it finds the property or method. This process continues until it reaches the `Object.prototype` at the top of the chain.

17. What is the difference between Call, Apply, and Bind methods?

Call: The `call()` method invokes a function with a specified `this` value and individual arguments passed as comma-separated values

```
const person1 = { name: 'John' };
const person2 = { name: 'Jane' };

function greet(greeting) {
  console.log(greeting + ' ' + this.name);
}

greet.call(person1, 'Hello'); // Output: Hello John
greet.call(person2, 'Hi'); // Output: Hi Jane
```

With `call()` method an object can use a method belonging to another object.

```
const o1 = {
  name: 'ravi',
  getName: function(){
    console.log(`Hello, ${this.name}`)
  }
}
```

```
const o2 = {  
  name: 'JavaScript Centric'  
}  
  
o1.getName.call(o2) // Hello, JavaScript Centric
```

Apply: Invokes the function with a given `this` value but it accepts arguments as an array. It is useful when the number of arguments to be passed is not known in advance or when the arguments are already in an array.

```
const numbers = [1, 2, 3, 4, 5];  
  
const max = Math.max.apply(null, numbers);  
console.log(max); // Output: 5
```

bind: instead of invoking it returns a new function and allows you to pass any number of arguments. `bind()` method takes an object as first argument and create a new function.


```
const module = {  
  x: 42,  
  getX: function() {  
    return this.x;  
  }  
};  
  
const boundGetX = unboundGetX.bind(module);  
console.log(boundGetX()); // Output: 42
```

18. What are lambda or arrow functions?

There are two types of functions in JavaScript

1. Regular Functions
2. Arrow Functions (Introduced in ES6)

Regular Function: We can write the regular function in two ways, i.e. **Function declaration**, and **Function expression**.




```
// Function declaration
function add(a, b) {
    return a + b;
}

// Function expression
const sum = function (a, b) {
    return a + b;
}

add(2, 3); // 5
sum(2, 3); // 5
```

Arrow or Fat Arrow Function: Lambda functions, also known as **arrow functions**, are a feature introduced in JavaScript (ES6) that is a more concise syntax for writing **function expressions**. They have a shorter syntax compared to traditional function expressions and are **particularly useful for creating anonymous functions** and working with functional programming concepts.

There is no declaration approach here, we can write by using Function expressions only.



```
// arrow function expression
const add = (a, b) => {
    return a + b;
}

// very simple and concise syntax
const add = (a, b) => a + b;
```


There are certain differences between Arrow and Regular function, i.e

1. Syntax
2. No arguments (arguments are array-like objects)
3. No prototype object for the **Arrow function**
4. Cannot be invoked with a new keyword (**Not a constructor function**)
5. No own this (**call, apply & bind won't work as expected**)
6. It cannot be used as a Generator function
7. Duplicate-named parameters are not allowed

19. What is the currying function?

Currying is a technique in functional programming that transforms a function with multiple arguments into a series of functions, each taking a single argument.

These curried functions can be composed together to build more complex functions.

In JavaScript, you can implement currying using closures and returning functions.

```
// Regular function taking two arguments
function add(x, y) {
  return x + y;
}
```

```
// Curried version of the function
function curryAdd(x) {
  return function(y) {
    return x + y;
  };
}

const add5 = curryAdd(5); // Partial application, creates a new function
console.log(add5(3)); // Output: 8
```

Currying is beneficial in functional programming and can make code more modular and reusable. It's particularly useful in scenarios where you want to create functions

with a varying number of arguments or build pipelines of data transformations.

20. What are the features of ES6?

ES6, also known as ECMAScript 2015, introduced several new features and enhancements to JavaScript, significantly expanding the language's capabilities. Some key features of ES6 include:

1. **Arrow Functions**
2. **Block-Scoped Variables**
3. **Classes**
4. **Modules**
5. **Template Literals:** Template literals allow for embedding expressions and multi-line strings using backticks, providing a more convenient way to create complex strings in JavaScript.
6. **Default Parameters**
7. **Rest and Spread Operators**
8. **Destructuring Assignment**
9. **Promises**
10. **Map, Set, WeakMap, WeakSet:** ES6 introduced new built-in data structures, such as Map and Set, for more efficient and specialized handling of collections and key-value pairs.
11. **Iterators and Generators**
12. **Enhanced Object Literals**

21. What is Execution context, execution stack, variable object, and scope chain?

Execution Context: the execution context refers to the environment in which a piece of code is executed. It consists of the scope, variable object, and the value of the "this" keyword.

Whenever a function is executed, an execution context is created and it contains all the variables or properties of that function.

There are three types of execution context in JavaScript:

> Global Execution Context

> Functional Execution Context

> Eval Function Execution Context

Execution Stack: It is also known as the “call stack,” a LIFO (Last in, First out) data structure that stores all the execution context of the function calls that are in progress. When a function is called, a new execution context is created and pushed onto the stack. When the function completes, its context is popped off the stack.

The engine executes the function whose execution context is at the top of the stack. When this function completes, its execution stack is popped off from the stack, and the control reaches the context below it in the current stack.

The execution context is created during the creation phase. The following things happen during the creation phase:

1. **LexicalEnvironment** component is created.
2. **VariableEnvironment** component is created.

Variable Object: It is a part of the execution context that contains all the variables, function declarations, and arguments defined in that context.

Scope Chain: The scope chain is a mechanism for resolving the value of a variable in JavaScript. When a variable is referenced, the JavaScript engine looks for the variable first in the current execution context's variable object. If it's not found there, it continues to the next outer execution context, following the scope chain, until it finds the variable or reaches the global execution context.

22. What is the priority of execution of callback, promise, setTimeout, process.nextTick()?

The priority of execution can be understood based on the event loop and the order in which different asynchronous operations are handled:

1. **process.nextTick():** Callbacks scheduled using `process.nextTick()` have the highest priority. When you use `process.nextTick()`, the callback is executed immediately after the current operation completes but before the event loop moves on to the next phase. This makes it a way to ensure that a function is executed at the earliest possible moment in the event loop.

2. **Promise:** Promises are typically executed after `process.nextTick()`. However, they are prioritized over callbacks scheduled with `setTimeout()`.
3. **setTimeout():** Callbacks scheduled with `setTimeout()` are placed in the timer phase of the event loop. They will be executed after the current operation, promises, and any previously scheduled `setTimeout()` callbacks have been completed.
4. **Callback:** Regular callbacks (not scheduled using `process.nextTick()`) have the lowest priority. They are executed after the event loop processes `process.nextTick()`, promises, and `setTimeout()` callbacks.

23. What is the Factory function and generator function?

A factory function in JavaScript is a function that returns an object. It is a pattern used to create objects in a straightforward and organized manner. Instead of using constructor functions and the new keyword to create new objects, a factory function encapsulates the object creation process and returns a new object.

```
function createPerson(name, age) {  
  return {  
    name: name,  
    age: age,  
    greet: function() {  
      return `Hello, my name is ${this.name} and I am ${this.age} years old.`;  
    }  
  };  
}  
  
const person1 = createPerson('Alice', 25);  
const person2 = createPerson('Bob', 30);  
  
console.log(person1.greet()); // Output: Hello, my name is Alice and I am 25 ye  
console.log(person2.greet()); // Output: Hello, my name is Bob and I am 30 year
```

A Generator function in JavaScript is a special type of function that can be paused and resumed during its execution.

A generator function produces a sequence of results instead of a single value.

When a generator function called **it returns a generator object** that can be used to control the execution of the function by calling the `next()` method.

The function's code can be paused within the body using the **yield keyword**, and it can later be resumed from the exact point where it was paused.

```
function* numberGenerator() {  
  let i = 0;  
  while (true) {  
    yield i++;  
  }  
}  
  
const gen = numberGenerator();  
console.log(gen.next().value); // Output: 0  
console.log(gen.next().value); // Output: 1  
console.log(gen.next().value); // Output: 2
```

This provides a powerful mechanism for creating iterators and handling asynchronous code.

24. Different ways to clone (Shallow and deep copy of object) an object?

A **shallow copy** is a copy of an object whose **references are the same** as the original object. This means that if you change the value of a property in the shallow copy, it will also change the value of the property in the original object.

```
const user = {  
  name: "Kingsley",  
  age: 28,  
  job: "Web Developer"  
}  
const clone = user
```

A **deep copy** is a copy of an object whose **references are not the same** as the original object. This means that if you change the value of a property in the deep copy, it will not change the value of the property in the original object.

there different ways to create deep copy of an object.

a) **JSON.parse** and **JSON.stringify**: useful for nested object also.

```
const originalObject = { name: "Alice", age: 25 };
const deepCopy = JSON.parse(JSON.stringify(originalObject));
```

b) **structuredClone**:

```
const myDeepCopy = structuredClone(myOriginal);
```

c) **Spread Operator(...)**: any object with a nested object will not be deep copied.

```
const originalObject = { name: "Alice", age: 25 };
const deepCopy = {...originalObject};

deepCopy.name = "ravi"
console.log("originalObject", originalObject.name) // Alice
```

d) **Object.assign()**: the `Object.assign()` method should be used to deep copy objects that have no nested objects.

```
const originalObject = { name: "Alice", age: 25 };
const shallowCopy = Object.assign({}, originalObject);
```

e) **Recursion**:

```
function deepCopy(obj) {
  if (typeof obj !== 'object' || obj === null) {
    return obj;
  }
  const newObj = Array.isArray(obj) ? [] : {};
  for (let key in obj) {
    if (Object.hasOwnProperty.call(obj, key)) {

```

```
        newObj[key] = deepCopy(obj[key]);
    }
}
return newObj;
}
const originalObject = { name: "Alice", nested: { age: 25 } };
const deepCopy = deepCopy(originalObject);
```

25. How to make an object immutable? (seal and freeze methods)?

In JavaScript, you can make an object immutable using the `Object.seal()` and `Object.freeze()` methods.

Object.freeze(): (Completely Immutable) this method freezes an object, making it both sealed and marking all its properties as read-only. After freezing an object, its properties cannot be modified, added, or removed.

```
const obj = { name: 'Alice', age: 25 };
Object.freeze(obj);
obj.name = 'Bob'; // Not allowed
obj.address = '123 Street'; // Not allowed
delete obj.age; // Not allowed
```

Object.seal(): (Partially Immutable) this method seals an object, preventing new properties from being added and marking all existing properties as non-configurable. However, you can still modify the values of existing properties that are writable.

```
const obj = { name: 'Alice', age: 25 };
Object.seal(obj);
obj.name = 'Bob'; // Allowed
obj.address = '123 Street'; // Not allowed (no new properties can be added)
delete obj.age; // Not allowed (existing properties cannot be deleted)
```

26. What is Event and event flow, event bubbling and event capturing?

In JavaScript, Event flow is the order in which an event like a click or a keypress is received on the web page or handled by the web browser. There are two phases in event flow: event capturing and event bubbling.

When you click an element that is nested in various other elements, before your click actually reaches its destination or target element, **it must trigger the click event for each of its parent elements first**, starting at the top with the global window object.

```
<div id="parent">
  <button id="child">Click me!</button>
</div>
```

Now, let's explain event flow with this example:

- 1. Event Capturing Phase:** When you click the button, the event starts its journey from the top (the root of the document) and moves down to the target element. In this case, it travels from the document's root to the `<div>` (parent element), then to the `<button>` (child element). This is called the capturing phase.
- 2. Event Target Phase:** The event reaches the target element, which is the `<button>` in this case.
- 3. Event Bubbling Phase:** After reaching the target, the event starts bubbling up. It goes from the `<button>` back to the `<div>` and eventually to the root of the document. This is called the bubbling phase.

Here's a simple JavaScript code snippet to see this in action:

```
document.getElementById('parent').addEventListener('click', function() {
  console.log('Div clicked (capturing phase)');
}, true); // The 'true' here indicates capturing phase.
```

```
document.getElementById('child').addEventListener('click',
function() {
  console.log('Button clicked (target phase)');
});

document.getElementById('parent').addEventListener('click',
function() {
  console.log('Div clicked (bubbling phase)');
});
```


When you click the button, you'll see these messages in the console in the following order:

1. "Div clicked (capturing phase)"
2. "Button clicked (target phase)"
3. "Div clicked (bubbling phase)"

27. What is Event delegation?

Event delegation is a JavaScript programming technique **that optimizes event handling for multiple elements.**

Instead of attaching an event listener to each individual element, event delegation involves attaching a single event listener to a common ancestor element that is higher up in the DOM (Document Object Model) hierarchy.

When an event occurs on one of the descendant elements, it "bubbles up" to the common ancestor, where the event listener is waiting.

Event delegation is a technique for listening to events where you delegate a parent element as the listener for all of the events that happen inside it.

```
var form = document.querySelector("#registration-form");
```

```
// Listen for changes to fields inside the form
form.addEventListener(
  "input",
  function (event) {
    // Log the field that was changed
    console.log(event.target);
  },
  false
);
```

28. What are server-sent events?

Server-sent events (SSE) are a simple and efficient technology for enabling **real-time updates from the server to the client over a single HTTP connection.**

SSE allows the server to push data to the web client (usually a browser) as soon as new information is available, making it an excellent choice for scenarios where you need real-time updates without relying on complex protocols or third-party libraries.

a) SSE provides a **unidirectional flow of data from the server to the client**. The server initiates the communication, sending updates to clients.

b) SSE **uses a text-based protocol**, which means that data sent from the server to the client is typically in a text format (usually JSON or plain text).

c) SSE handles reconnection automatically.

d) SSE establishes a persistent connection between the client and the server, allowing the server to send a stream of events to the client. Each event can have a unique type and data associated with it.

e) **The EventSource object is used to receive server-sent event notifications**. For example, you can receive messages from server as below,

```
if (typeof EventSource !== "undefined") {  
  var source = new EventSource("sse_generator.js");  
  source.onmessage = function (event) {  
    document.getElementById("output").innerHTML += event.data + "<br>";  
  };  
}
```

f) Below are the list of events (onopen, onmessage, onerror) available for server-sent events.

29. What is a web worker or service worker in javascript?

Web Workers and Service Workers are two different concepts in JavaScript,

Web Workers are designed for concurrent JavaScript execution in the background, while **Service Workers are used for creating Progressive Web Apps with offline capabilities** and advanced features. Both are essential tools for enhancing the performance and functionality of web applications.

Each serves a distinct purpose in web development:

Web Workers:

1. **Concurrency:** Web Workers are a browser feature that allows you to run JavaScript code in the background, **separate from the main browser thread**. This enables concurrent execution of tasks without blocking the user interface.
2. **Use Cases:** Web Workers are commonly used for tasks that are computationally intensive or time-consuming, such as data processing, image manipulation, or complex calculations. By running these tasks in a separate thread, they don't impact the responsiveness of the web page.
3. **Communication:** Web Workers can **communicate with the main thread using a messaging system**. They can send and receive messages, allowing for coordination between the main thread and the worker.
4. **Browser Support:** Web Workers are supported in most modern browsers.

Service Workers:

1. **Offline Capabilities:** Service Workers are a more advanced feature used for creating Progressive Web Apps (PWAs). They act as proxy servers that run in the background and can intercept and cache network requests. This enables offline capabilities, such as serving cached content when the user is offline.
2. **Use Cases:** Service Workers are primarily used for implementing features like **offline access, push notifications, and background sync**. They enable web apps to function even when there's no internet connection.
3. **Lifecycle:** Service Workers have their own lifecycle with events like `install`, `activate`, and `fetch`. They are typically registered at the beginning of a web app's life.
4. **Browser Support:** Service Workers are supported in modern browsers and are a key technology for creating reliable and engaging web applications.

30. How to compare 2 JSON objects in javascript?

a) One simple way to compare two JSON objects is to use `JSON.stringify` to convert them into strings and then compare the strings.

```
function areEqual(obj1, obj2) {  
  return JSON.stringify(obj1) === JSON.stringify(obj2);  
}
```

```
const obj1 = { a: 1, b: { c: 2 } };
const obj2 = { a: 1, b: { c: 2 } };
console.log(areEqual(obj1, obj2)); // Output: true
```

b) You can use the Ramda library to compare two JSON objects as well. Ramda provides a function called `equals` for this purpose.

```
const R = require('ramda');

const obj1 = { a: 1, b: { c: 2 } };
const obj2 = { a: 1, b: { c: 2 } };

console.log(R.equals(obj1, obj2)); // Output: true
```

c) Another option is to use a library, such as Lodash, that provides a method for deep comparison of objects.

```
const _ = require('lodash');

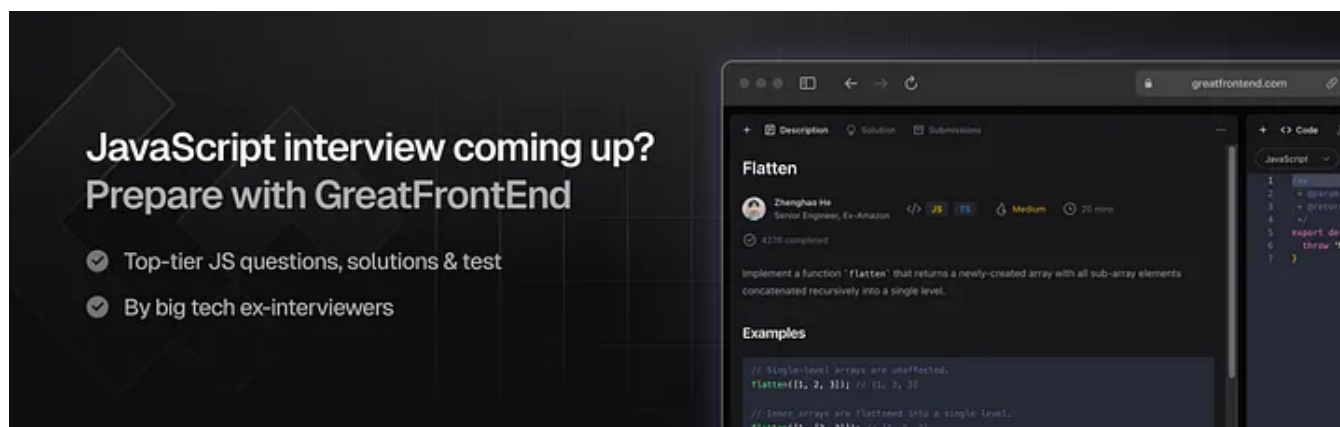
const obj1 = { a: 1, b: { c: 2 } };
const obj2 = { a: 1, b: { c: 2 } };
console.log(_.isEqual(obj1, obj2)); // Output: true
```

Hope You Like this article. **Big Thank you For Reading.**

Follow me on Medium for more guided articles.

Follow me on LinkedIn: <https://www.linkedin.com/in/ravics09/>

. . .



💡 Nail JavaScript interviews with the right practice questions and in-depth solutions from ex-interviewers! [Try GreatFrontEnd](#) → 💡

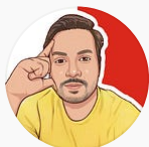
JavaScript

Javascript Tips

Javascript Development

Reactjs

Nodejs



Follow



Written by Ravi Sharma

1.2K Followers

JavaScript Developer | <https://www.youtube.com/channel/UC9MmyicGlveu0AId8OFAOmQ>

More from Ravi Sharma