**1.what is hoisting in javascript**  
  
In JavaScript, **hoisting** refers to the behavior where variable and function declarations are moved, or "hoisted," to the top of their containing scope (either global or function scope) before the code is executed. This means that you can use variables and functions before they are declared in the code.  
  
However, **only the declarations are hoisted**, not the initializations (for variables). This can sometimes lead to unexpected behavior if not fully understood.  
  
**Function Hoisting**: Function declarations are hoisted completely, which means you can call a function before its declaration in the code.  
  
  
Example code: // Works fine, function is hoisted

hello();

function hello() {

console.log('Hello, World!');

}

**Variable Hoisting**: Variables declared with var are hoisted but **not initialized**. The declaration is hoisted, but the initialization remains in place. Until the initialization is encountered in the code, the variable is undefined.

// undefined (because declaration is hoisted, but not the assignment)

console.log(a);

var a = 5;

console.log(a);

For let and const, hoisting still occurs, but they are in a **temporal dead zone** from the start of the block until the declaration is encountered, which means trying to access them before declaration will throw an error.

// ReferenceError: Cannot access 'b' before initialization   
console.log(b);

let b = 10;

console.log(b)

Let and const are for block it throw Reference error  
For Var it check global

Since it is global scope variable by default its is undefined if you won’t assign anything

### **Summary:**

* Function declarations are fully hoisted.
* Variable declarations (var) are hoisted but not their assignments.
* Variables declared with let and const are hoisted, but they cannot be accessed before their declaration due to the temporal dead zone.

**2. what is memoization?**

Memoization is an optimization technique that stores the results of expensive function calls and returns the cached result when the same inputs occur again. By caching the results, it avoids the need to recompute the same values, improving performance.

In JavaScript, memoization is often implemented using a higher-order function that wraps another function and stores the results in a cache object.

function memoize(fn) {

const cache = {};

return function(...args) {

const key = JSON.stringify(args);

if (cache[key]) {

console.log('Cached data')

return cache[key]; // Return the cached result

}

const result = fn(...args); // Compute the result

cache[key] = result; // Store the result in the cache

return result;

};

}

// Example usage:

const slowFunction = (num) => {

console.log('Computing...');

return num \* 2;

};

const memoizedFunction = memoize(slowFunction);

console.log(memoizedFunction(5)); // Output: Computing... 10

console.log(memoizedFunction(5)); // Output: 10 (cached result, no "Computing..." log)

console.log(memoizedFunction(4));

console.log(memoizedFunction(4));

console.log(memoizedFunction(5));

**3. .Javascript is synchronous or asynchronous.**

1. **Synchronous Execution**:
   * JavaScript runs code sequentially, so each statement must complete before the next one begins.
   * This can lead to blocking behavior, where long-running operations (like fetching data from a server) can pause other code from executing.
2. **Asynchronous Execution**:
   * JavaScript can handle asynchronous tasks using callbacks, promises, and async/await syntax.
   * This allows for non-blocking behavior, meaning other code can run while waiting for an asynchronous operation to complete.

### **Common Asynchronous Patterns:**

* **Callbacks**: Functions that are passed as arguments to be executed later.
* **Promises**: Objects representing the eventual completion (or failure) of an asynchronous operation and its resulting value.
* **Async/Await**: Syntactic sugar built on promises that makes asynchronous code easier to read and write.

console.log("Start");

setTimeout(() => {

console.log("Timeout completed");

}, 2000);

console.log("End");

**4. Promise vs async await**:  
  
Both Promise and async/await are used in JavaScript to handle asynchronous operations, but they work in different ways. Here's a comparison between them:

1. A Promise is an object representing the eventual completion or failure of an asynchronous operation. It allows you to attach callbacks using .then() for success and .catch() for errors.

function fetchData() {

return new Promise((resolve, reject) => {

setTimeout(() => {

const success = true;

if (success) {

resolve("Data received");

} else {

reject("Error occurred");

}

}, 1000);

});

}

fetchData()

.then(data => {

console.log(data); // "Data received"

})

.catch(error => {

console.error(error); // "Error occurred"

});

### 

1. async/await is syntactic sugar built on top of Promises. It allows you to write asynchronous code in a more synchronous and readable manner. An async function automatically returns a Promise, and await is used to wait for a Promise to resolve.

async function fetchData() {

try {

const data = await new Promise((resolve, reject) => {

setTimeout(() => {

const success = true;

if (success) {

resolve("Data received");

} else {

reject("Error occurred");

}

}, 1000);

});

console.log(data); // "Data received"

} catch (error) {

console.error(error); // "Error occurred"

}

}

fetchData();

### **Key Differences:**

1. **Readability**:
   * async/await makes asynchronous code look more like synchronous code, improving readability, especially when chaining multiple promises.
   * Promises require handling with .then() and .catch(), which can become less readable when there are multiple chains.
2. **Error Handling**:
   * With async/await, you can use try/catch blocks for error handling, making it more intuitive.
   * Promises handle errors via .catch().
3. **Execution Flow**:
   * await pauses the function execution until the promise resolves, giving a synchronous appearance. Promises, on the other hand, don’t pause but return immediately, making it harder to follow the execution flow.
4. **Compatibility**:
   * Promises work in older JavaScript environments, while async/await was introduced in ES2017 (ES8), so some older environments might need polyfills or transpiling.

### **When to Use Which?**

* Use async/await when you want more readable and synchronous-looking asynchronous code.
* Use Promises when you're dealing with concurrent promises or need to chain them (e.g., Promise.all(), Promise.race()), or when you don’t want to wait for one promise before moving on to others.

In modern development, async/await is generally preferred for its readability, but Promises are still used when specific control over promise behavior is needed.  
  
  
In JavaScript, Promise.all(), Promise.race(), and Promise.any() are methods that work with multiple promises at once, allowing you to handle multiple asynchronous operations in parallel. Here's how each one works:

**5. what is promise.all, promise.race, promise.any**

### **1. Promise.all()**

* **What it does**: Takes an array (or any iterable) of promises and returns a single promise that resolves when **all** of the promises have resolved.
* **Behavior**:
  + If all promises resolve successfully, the returned promise resolves with an array of the resolved values.
  + If any promise is rejected, the returned promise will reject with that reason.

const promise1 = Promise.resolve(10);

const promise2 = Promise.resolve(20);

const promise3 = Promise.resolve(30);

Promise.all([promise1, promise2, promise3])

.then(values => {

console.log(values); // Output: [10, 20, 30]

})

.catch(error => {

console.log(error); // This runs if any promise is rejected

});

### **2. Promise.race()**

* **What it does**: Takes an array of promises and returns a single promise that resolves or rejects as soon as **one** of the promises in the array settles (resolves or rejects).
* **Behavior**:
  + The returned promise settles with the result (resolve/reject) of the first promise to complete, regardless of whether it resolves or rejects.

const promise1 = new Promise((resolve) => setTimeout(resolve, 500, 'First'));

const promise2 = new Promise((resolve) => setTimeout(resolve, 100, 'Second'));

Promise.race([promise1, promise2])

.then(value => {

console.log(value); // Output: "Second" because promise2 resolves first

});

**3. Promise.any()**

* **What it does**: Takes an array of promises and returns a single promise that resolves as soon as **any one** of the promises resolves. It ignores rejections and only rejects if **all** promises are rejected.
* **Behavior**:
  + The returned promise resolves with the first successfully resolved promise.
  + If all promises reject, it rejects with an AggregateError containing all the rejection reasons.

const promise1 = new Promise((\_, reject) => setTimeout(reject, 100, 'Error 1'));

const promise2 = new Promise((resolve) => setTimeout(resolve, 200, 'Success'));

const promise3 = new Promise((\_, reject) => setTimeout(reject, 300, 'Error 2'));

Promise.any([promise1, promise2, promise3])

.then(value => {

console.log(value); // Output: "Success"

})

.catch(error => {

console.log(error); // Only runs if all promises reject

});

### **Summary:**

* **Promise.all()**: Waits for all promises to resolve, and fails if any of them reject.
* **Promise.race()**: Resolves or rejects as soon as any promise resolves or rejects.
* **Promise.any()**: Resolves as soon as any promise resolves, and only fails if all promises reject.

6. What is destructuring in JavaScript

Destructuring is a syntax feature in JavaScript (and other programming languages) that allows you to unpack values from arrays or properties from objects into distinct variables. It makes code cleaner and easier to read by reducing the need for repetitive access to array elements or object properties.

**Program question:**

for(var i=0, i<5, i++){  
setTimeOut(()=>{

console.log(i);

}, 1000)

}

**With using var display 0,1,2,3**

for (var i = 0; i < 5; i++) {

(function(i) {

setTimeout(() => {

console.log(i);

}, 1000);

})(i);

}

**Create an array of objects which has age, name, marks. Get me the names of those whose marks is greater than 65**

const students = [

{ age: 20, name: "Alice", marks: 70 },

{ age: 22, name: "Bob", marks: 65 },

{ age: 19, name: "Charlie", marks: 80 },

{ age: 21, name: "David", marks: 60 },

{ age: 23, name: "Eve", marks: 90 }

];

// Get names of students whose marks are greater than 65

const namesWithHighMarks = students

.filter(student => student.marks > 65)

.map(student => student.name);

console.log(namesWithHighMarks);

**[0,1,0,0,1,1,0,1]**

**Move all zeros to right**

function moveZerosToRight(arr) {

// Create an array to hold non-zero elements

const nonZeroElements = arr.filter(num => num !== 0);

// Count the number of zeros in the original array

const zeroCount = arr.length - nonZeroElements.length;

// Create a new array with non-zero elements followed by the zeros

const result = [...nonZeroElements, ...Array(zeroCount).fill(0)];

return result;

}

// Example usage:

const inputArray = [0, 1, 0, 0, 1, 1, 0, 1];

const outputArray = moveZerosToRight(inputArray);

console.log(outputArray);

**[0,1,0,0,1,1,0,1]**

**Move all zeros to right without using an extra array**

function moveZerosToRight(arr) {

let lastNonZeroIndex = 0;

// Loop through the array

for (let i = 0; i < arr.length; i++) {

// If the current element is not zero, move it to the lastNonZeroIndex

if (arr[i] !== 0) {

arr[lastNonZeroIndex] = arr[i];

lastNonZeroIndex++;

}

}

// After all non-zero elements have been moved, fill the rest of the array with zeros

for (let i = lastNonZeroIndex; i < arr.length; i++) {

arr[i] = 0;

}

}

// Example usage:

const inputArray = [0, 1, 0, 0, 1, 1, 0, 1];

moveZerosToRight(inputArray);

console.log(inputArray); // Output: [1, 1, 1, 1, 0, 0, 0, 0]