Feature	var	let	const
Scope	**Function-scoped**	**Block-scoped**	**Block-scoped**
Redeclare	✅ Allowed	❌ Not allowed	❌ Not allowed
Reassign	✅ Allowed	✅ Allowed	❌ Not allowed
Hoisting	✅ Hoisted (initialized as `undefined`)	✅ Hoisted (but in *temporal dead zone*)	✅ Hoisted (but in *temporal dead zone*)
Use Case	Old code / function scope	Modern variable	Fixed values / constants

Hoisting

- All are **hoisted** (moved to the top of scope before code runs).
- But `var` gets initialized as `undefined`, while `let`/`const` are in **Temporal Dead Zone (TDZ)** until actual declaration line is reached.

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

console.log(y); // ❌ ReferenceError (TDZ)
let y = 5;
```

What is the Temporal Dead Zone?
A:
The time between entering the scope and the variable being declared is called the Temporal Dead Zone (TDZ). Variables in TDZ cannot be accessed before declaration, causing a **ReferenceError**.

Can you change the value of a `const` object?
A:
Yes, you can change properties of a `const` object — the reference is constant, not the content.

```
const obj = { name: "John" };
obj.name = "Doe"; // ✅ allowed
// obj = { age: 30 }; ❌ Error (changing reference)
```

Hoisting in JavaScript means:

> During the execution phase, **variable and function declarations** are moved ("hoisted") to the **top of their scope** before the code runs.

This means you can **use a function or variable before it appears in your code** — but the behavior differs between `var`, `let`, and `const`.

How it Works

When JavaScript runs your code, it does two main things:

1. **Creation phase** – allocates memory for variables and functions.
2. **Execution phase** – runs the code line-by-line.

In the creation phase:

- **Function declarations** are hoisted **with their body**.

- **var variables** are hoisted **and initialized as `undefined`**.
- **let/const variables** are hoisted but **not initialized** → they stay in the **Temporal Dead Zone (TDZ)** until their declaration line is reached.

Function Hoisting

```
sayHi(); // ✅ Works — hoisted completely
function sayHi() {
  console.log("Hi");
}
```

** But **function expressions** behave like variables:

```
sayHello(); // ❌ TypeError: sayHello is not a function
var sayHello = function() {
  console.log("Hello");
};
```

var → hoisted, initialized as `undefined`.

- **let/const** → hoisted, but **in TDZ** until declaration line.
- **Function declarations** → fully hoisted.
- **Function expressions** → behave like variables.

What is the difference between function declaration and function expression in hoisting?
A: Function declarations are hoisted completely (you can call them before they are written), while function expressions are hoisted like variables (initialized as `undefined`).

Normal Functions (Function Declarations & Expressions)

Function Declaration

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

console.log(add(2, 3)); // 5
```

Hoisted completely → you can call it before definition.

- Has its own `this` context.
- Can use the `arguments` object.

Function Expression

```
const multiply = function(a, b) {
  return a * b;
};

console.log(multiply(4, 5)); // 20
```

Assigned to a variable.

- Hoisted like a variable → **cannot call before definition**.
- Still has its own `this`.

Arrow Functions
const subtract = (a, b) => a - b;

console.log(subtract(5, 2)); // 3

Key Features:

- Shorter syntax.
- **No own `this`** → inherits `this` from surrounding scope (**lexical this**).
- **No `arguments` object** → need rest parameters if needed.
- Cannot be used as a constructor (`new`).

Feature	Normal Function	Arrow Function
Syntax	Longer	Short & concise
`this` binding	Own `this` (dynamic)	Lexical `this` (from parent scope)
`arguments`	✅ Available	❌ Not available
Constructor (`new`)	✅ Allowed	❌ Not allowed
Hoisting	Declaration hoisted	Expression hoisted like variable
Best Use	Complex logic, methods needing `this`	Short callbacks, functional programming

Function declaration → hoisted, has own `this`.

- Function expression → not hoisted fully, still has own `this`.
- Arrow function → no own `this` or `arguments`, shorter syntax, good for callbacks.
- Avoid arrow functions for **object methods** where `this` is needed.

**

Rule of `this`

- In **normal functions**, `this` is **dynamic** — it's decided **by how the function is called**.
- In **arrow functions**, `this` is **lexical** — it's decided **by where the function is written**, not how it's called.
  (Arrow functions don't have their own `this`; they use the one from the surrounding scope.)

```
const obj1 = {
  name: "Abhishek",
  sayName: function() {
    console.log(this.name);
  }
};
```

obj1.sayName(); // "Abhishek"
`sayName` is a **normal function**.

- When you call `obj1.sayName()`, the **caller** is `obj1`.
- So, `this` points to `obj1`.
- `this.name` → `"Abhishek"` ✅.

```
const obj2 = {
  name: "Abhishek",
  sayName: () => {
    console.log(this.name);
  }
};
obj2.sayName(); // undefined
```

`sayName` is an **arrow function**.

- Arrow functions **don't have their own `this`**.
- So they look for `this` **in the place where the function is defined**.
- Here, `sayName` is defined **inside the object literal**, but the surrounding scope is actually the **global scope** (or module scope), **not** `obj2`.
- In the global scope, `this` is:
- `window` (in browsers) → has no `name` property.
- `undefined` in strict mode.
- So, `this.name` → `undefined`.

Think of `this` as "**who owns the function call**":

- **Normal function:**
  If `obj1.sayName()` → **obj1 owns it**, so `this = obj1`.
- **Arrow function:**
  It doesn't care who calls it — it remembers `this` from when it was **created**.

```
const obj = {
  name: "Abhishek",
  normalFn: function() {
    console.log("Normal:", this.name);
  },
  arrowFn: () => {
    console.log("Arrow:", this.name);
  }
};

obj.normalFn(); // Normal: Abhishek
obj.arrowFn();  // Arrow: undefined
```

```
// Now let's store them in variables
const n = obj.normalFn;
const a = obj.arrowFn;

n(); // Normal: undefined (this is now global, not obj)
a(); // Arrow: undefined (still lexical from where it was made)
```

**

case where `this` in an **arrow function** is **not** `undefined`.
This happens when the **outer scope** where the arrow function is created already has a valid `this`.

```
const obj = {
  name: "Abhishek",
  sayName: function() {
    // Normal function → here 'this' is obj
    const arrow = () => {
      console.log(this.name); // 'this' from sayName's scope
    };
    arrow();
  }
};

obj.sayName(); // Abhishek
```

`sayName` is a **normal function**, so `this` = `obj`.

- The **arrow function** `arrow` is created inside `sayName`.
- Arrow functions don't have their own `this`, so they use the **`this` from where they were created** — which is `obj` here.

```
const obj = {
  name: "Abhishek",
  greet: function() {
    setTimeout(() => {
      console.log("Hello", this.name);
    }, 1000);
  }
};

obj.greet(); // After 1s → Hello Abhishek
```

`greet` is a normal method → `this` = `obj`.

- Arrow function inside `setTimeout` **inherits** that same `this`.

if you **declare an arrow function at the top level** — meaning **not inside another function** — then `this` will **not** point to anything useful.

CALLBACK FUNCTIONS

What is a Callback Function?

A **callback function** is just a function that you **pass as an argument** to another function so that the other function can **call it later**.They are often used for asynchronous operations.

Why use a callback instead of just calling a function directly?
A: Because you may not want to run it right away — callbacks let the other function decide *when* and *if* it should run, useful in async tasks and reusable code.

What's the problem with callbacks?
A: If nested too deeply, they cause **callback hell** — making code hard to read and maintain. This is why promises and `async/await` were introduced.

```
function processUser(name, callback) {

  console.log("Processing user: " + name);

  callback(name); // flexible

}


function greet(name) {

  console.log("Hello " + name);

}


function goodbye(name) {

  console.log("Goodbye " + name);

}


processUser("Abhishek", greet);    // Hello Abhishek

processUser("Abhishek", goodbye);  // Goodbye Abhishek
```

Promises

```
getUser(id, (user) => {
  getOrders(user.id, (orders) => {
```

```
        getProducts(orders, (products) => {
            console.log(products);
        });
    });
});

getUser(id)
    .then(user => getOrders(user.id))
    .then(orders => getProducts(orders))
    .then(products => console.log(products))
    .catch(err => console.error(err));
```

What is a Promise?

A **Promise** is an object that represents the **eventual result** (or failure) of an asynchronous operation.

Promise States

A Promise can be:

1. **pending** → operation is still running
2. **fulfilled** → operation finished successfully
3. **rejected** → operation failed

```
const mypromise = new Promise((resolve,reject)=>{
    setTimeout(() => {
      const success = false
      if (success) {
       resolve("Data loaded")
      }else{
       reject("Data loading failed")
      }
    }, 1000);
})

mypromise.then((data)=>{
    console.log(data)
}).catch((error)=>{
    console.log(error)
}).finally(()=>{
    console.log("Finally block")
})
```

How do Promises solve the problem of callback hell?
A: Promises allow you to attach `.then()` handlers in a flat chain instead of deeply nested callbacks. They also centralize error handling using `.catch()`.

response.json(): Parses JSON response from the server to a JavaScript object (often used with fetch()). • JSON.stringify(): Converts a JavaScript object into a JSON-formatted string. • JSON.parse(): Converts a JSON-formatted string back into a JavaScript object

await: This keyword is used to pause the execution of an async function until the Promise resolves (or rejects). It can only be used inside functions marked as async.

```
async function main() {
  try {
    const user = await getUser(id);
    const orders = await getOrders(user.id);
    const products = await getProducts(orders);
    console.log(products);
  } catch (error) {
    console.error(error);
  }
}
----------------------
```

```
// Basic object destructuring
const user = { name: 'John', age: 30, id: 123 };
const { name, age } = user;
console.log(name); // 'John'
console.log(age);  // 30

// Nested destructuring
const user = {
  name: 'John',
  address: { city: 'NY', country: 'USA' }
};
const { address: { city } } = user;
console.log(city); // 'NY'

Example:
cy.request('GET', '/api/user/1').then(({ body, status }) => {
  expect(status).to.eq(200);
  expect(body.name).to.eq('John');
});
```

Array Dest..

// Basic array destructuring
const numbers = [1, 2, 3];
const [first, second] = numbers;
console.log(first); // 1
console.log(second); // 2

// Skipping items
const [first, , third] = numbers;
console.log(third); // 3

// Rest pattern
const [first, ...rest] = numbers;
console.log(rest); // [2, 3]

```
// Default values
const [a = 1, b = 2] = [10];
console.log(a); // 10
console.log(b); // 2 (default)
```

Spread Operator

```
// Arrays
const arr1 = [1, 2, 3];
const arr2 = [...arr1, 4, 5]; // [1, 2, 3, 4, 5]

// Objects
const obj1 = { a: 1, b: 2 };
const obj2 = { ...obj1, c: 3 }; // { a: 1, b: 2, c: 3 }

// Function arguments
const numbers = [1, 2, 3];
Math.max(...numbers); // 3
```

Optional Chaining:

```
const user = { profile: { name: 'John' } };
console.log(user?.profile?.name); // 'John'
console.log(user?.address?.city); // undefined (no error)
```

** Array methods

```
// find() - useful in Cypress
const users = [{ id: 1, name: 'John' }, { id: 2, name: 'Jane' }];
const user = users.find(u => u.id === 2); // { id: 2, name: 'Jane' }

// includes()
[1, 2, 3].includes(2); // true

// map() - common in test data generation
const numbers = [1, 2, 3];
const doubled = numbers.map(n => n * 2); // [2, 4, 6]
```

** Closures in Javascript

Inner method remember the outside method properties

```
function outer() {
  const outerVar = "I'm outside!";

  function inner() {
    console.log(outerVar); // Accesses outerVar even after outer() finishes
  }

  return inner;
}

const myClosure = outer(); // outer() has finished executing
myClosure(); // Logs: "I'm outside!" (still remembers outerVar)
```

MAP FILTER REDUCE

```
//map
const arr = [-3,-5,-2,-3]
//square
console.log(arr.map((n)=>n*n));
// double
console.log(arr.map((n)=>2*n));
//binary
console.log(arr.map((n)=>n.toString(2)));

// filter

const array =[4,7,6,5,9];
//filter odd values
console.log(array.filter((n)=>n%2==1));
//filter even values
console.log(array.filter((n)=>n%2==0));

//reduce

//sum or max
function sum(arr){
    let sum=0;
    for(let ele of arr){
        sum += ele;
    }
    return sum;
}

console.log(sum(arr));

//using reduce
//acc is like sum and curr is the elemts iterated through
//0 is the intial val of acc
console.log(arr.reduce((acc,curr)=>{
    acc += curr;
    return acc;
},0));

//max

console.log(arr.reduce((acc,curr)=>{
    acc = Math.max(acc,curr);
    return acc;
},-Infinity));

//

const myArray = [3,2,5,8,2];
const mySum = myArray.reduce((acc,curr)=>{
    acc += curr;
```

```
  return acc;
},0);

console.log(mySum);
```

-------------------------UTILS-------------------------

Method	What it does	Example
`push()`	Adds element(s) **to the end** of array, returns new length.	`arr.push(4)` → `[1, 2, 3, 4]`
`pop()`	Removes **last element**, returns it.	`arr.pop()` → returns `3` from `[1, 2, 3]`
`unshift()`	Adds element(s) **to start**, returns new length.	`arr.unshift(0)` → `[0, 1, 2]`
`shift()`	Removes **first element**, returns it.	`arr.shift()` → returns `1` from `[1, 2]`
`concat()`	Joins arrays into new one.	`[1,2].concat([3,4])` → `[1,2,3,4]`
`join()`	Joins elements into a string.	`['a','b'].join('-')` → `"a-b"`
`slice(start, end)`	Copies part of array (non-mutating).	`arr.slice(1,3)`
`splice(start, deleteCount, ...items)`	Add/remove elements **in place**.	`arr.splice(1,1,'x')`
`indexOf()`	Finds index of element (or `-1`).	`arr.indexOf('apple')`
`includes()`	Checks if element exists (boolean).	`arr.includes(5)`
`find(fn)`	Finds first element matching condition.	`arr.find(x => x>3)`
`findIndex(fn)`	Finds index of first match.	`arr.findIndex(x => x>3)`
`filter(fn)`	Returns new array with matches.	`arr.filter(x => x>3)`
`map(fn)`	Returns new array after transforming each item.	`arr.map(x => x*2)`
`reduce(fn, initial)`	Reduces array to a single value.	`arr.reduce((a,b)=>a+b,0)`
`some(fn)`	Returns true if **any** match condition.	`arr.some(x=>x<0)`
`every(fn)`	Returns true if **all** match condition.	`arr.every(x=>x>0)`
`sort(fn)`	Sorts array (mutates).	`arr.sort((a,b)=>a-b)`
`reverse()`	Reverses array in place.	`arr.reverse()`
`flat(depth)`	Flattens nested arrays.	`[1,[2,3]].flat()`
`flatMap(fn)`	Map + flatten in one step.	`arr.flatMap(x=>[x,x*2`

Method	What it does	Example
		])

StrinG

Method	What it does	Example
`length`	Number of characters.	`"abc".length → 3`
`charAt(index)`	Character at index.	`"abc".charAt(1) → "b"`
`charCodeAt(index)`	Unicode value at index.	`"A".charCodeAt(0) → 65`
`at(index)`	Character at index (supports negative).	`"abc".at(-1) → "c"`
`indexOf()`	Finds position of substring.	`"hello".indexOf("e") → 1`
`lastIndexOf()`	Finds last occurrence.	`"banana".lastIndexOf("a")`
`includes()`	Checks if contains substring.	`"hello".includes("he")`
`startsWith()`	Checks start match.	`"hello".startsWith("he")`
`endsWith()`	Checks end match.	`"hello".endsWith("lo")`
`slice(start, end)`	Extracts part of string.	`"hello".slice(1,3) → "el"`
`substring(start, end)`	Like slice but no negative index.	`"hello".substring(1,3)`
`substr(start, length)`	Extracts part by length (**deprecated**).	`"hello".substr(1,2)`
`toUpperCase()`	All caps.	`"hi".toUpperCase()`
`toLowerCase()`	All lowercase.	`"HI".toLowerCase()`
`trim()`	Removes spaces both sides.	`" hi ".trim()`
`trimStart()` / `trimEnd()`	Removes spaces from start/end.	`" hi".trimStart()`
`padStart(len, str)`	Pads at start to length.	`"5".padStart(3,"0") → "005"`
`padEnd(len, str)`	Pads at end to length.	`"5".padEnd(3,"0") → "500"`
`repeat(n)`	Repeats string.	`"ha".repeat(3) → "hahaha"`
`replace(find, new)`	Replaces first match.	`"hi hi".replace("hi","bye")`
`replaceAll(find, new)`	Replaces all matches.	`"hi hi".replaceAll("hi","bye")`
`split(sep)`	Splits into array.	`"a-b-c".split("-")`
`match(regex)`	Returns regex matches.	`"abc123".match(/\d+/)`
`matchAll(regex)`	Returns all regex matches (iterator).	`[...str.matchAll(/a./g)]`
`search(regex)`	Returns index of regex match.	`"abc".search(/b/)`

OOPS

Classes

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

  greet() {
    console.log(`Hello, I'm ${this.name}!`);
  }
}

const alice = new Person("Alice", 25);
alice.greet(); // "Hello, I'm Alice!"
```

1. Encapsulation

- Bundling data (properties) and methods (functions) inside a class.
- **Private fields** (`#`) restrict direct access.

```
class BankAccount {
  #balance = 0; // Private field

  deposit(amount) {
    this.#balance += amount;
  }

  getBalance() {
    return this.#balance;
  }
}

const account = new BankAccount();
account.deposit(100);
console.log(account.getBalance()); // 100
// console.log(account.#balance); ❌ Error (private)
```

Inheritance

- A **child class** inherits properties/methods from a **parent class**.
- Uses `extends` keyword.

```
class Animal {
  constructor(name) {
```

```
    this.name = name;
  }

  speak() {
    console.log(`${this.name} makes a sound.`);
  }
}

class Dog extends Animal {
  speak() {
    console.log(`${this.name} barks!`);
  }
}

const dog = new Dog("Rex");
dog.speak(); // "Rex barks!"
```

Polymorphism

•A method behaves differently based on the object calling it.

•Achieved via **method overriding**.

```
class Bird extends Animal {
  speak() {
    console.log(`${this.name} chirps!`);
  }
}

const bird = new Bird("Tweety");
bird.speak(); // "Tweety chirps!"
```

Abstraction

•Hiding complex logic, exposing only necessary features.

•Uses **abstract classes/interfaces** (TypeScript supports this better).

```
class Car {
  #startEngine() { // Private method
    console.log("Engine started.");
  }

  drive() {
    this.#startEngine();
    console.log("Car is moving.");
```

```
  }
}

const car = new Car();
car.drive(); // "Engine started. Car is moving."
// car.#startEngine(); ❌ Error (private)
```

Static Methods & Properties

•Belongs to the **class**, not instances.

•Called using the class name.

What are the key principles of OOP in JavaScript?

Answer:

The **4 pillars of OOP** in JavaScript are:

Principle	Definition	Example
Encapsulation	Bundling data + methods in a class, hiding internal details	`class BankAccount { #balance = 0; }`
Inheritance	Child classes inherit properties/methods from parents	`class Dog extends Animal {}`
Polymorphism	Same method behaves differently in different classes	`animal.speak()` → `dog.speak()` barks, `cat.speak()` meows
Abstraction	Exposing only essential features, hiding complexity	Private methods (`#startEngine()`)

---

2. How do you create private variables in JavaScript?

Use `#` (hash prefix) for private fields (ES2022)

Prototype
```
function Animal(name) {
  this.name = name;
}

Animal.prototype.speak = function() {
  console.log(`${this.name} makes a noise.`);
};

const dog = new Animal("Rex");
dog.speak(); // "Rex makes a noise."
```

Concept	JavaScript Support	Example
Overriding (Same method name, different implementation in child class)	✅ Supported	`class Dog extends Animal { speak() { ... } }`
Overloading (Same method name, different parameters)	❌ Not supported (use default params)	`function greet(name, age = 0) { ... }`

. What are getters/setters? Why use them?

Answer:

•**Getters**: Control **read** access to properties.

•**Setters**: Control **write** access to properties.

=====
Singleton Class:

```
class Database {
  static #instance; // Private static field

  constructor() {
    if (Database.#instance) {
      return Database.#instance;
    }
    Database.#instance = this;
  }
}

const db1 = new Database();
const db2 = new Database();
console.log(db1 === db2); // true (same instance)
```

Difference between == and ===

Answer:

- `==` → Loose equality (performs type coercion).
- 5 == '5' // true

`===` → Strict equality (no type coercion, type must match).
5 === '5' // false

What is event bubbling and event capturing?

Answer:

- **Event Bubbling**: Event propagates from the target element **upwards** to the root (`document`).
- **Event Capturing**: Event propagates from the root **downwards** to the target.

What is hoisting in JavaScript?

Answer:

- Variable and function declarations are moved to the top of their scope **during compilation**.
- Variables declared with `var` are hoisted but initialized as `undefined`.
- `let` and `const` are hoisted but remain in the **Temporal Dead Zone** until declared.

Explain closures with an example

Answer:
A closure is when a function "remembers" variables from its **outer scope**, even after the outer function has finished.

```
function outer() {
  let count = 0;
  return function inner() {
    count++;
    return count;
  };
}
const counter = outer();
counter(); // 1
counter(); // 2
```

Difference between synchronous and asynchronous code

Answer:

- **Synchronous**: Code runs line-by-line, each operation must finish before moving to the next.
- **Asynchronous**: Code can run while waiting for other tasks (e.g., API calls, timers) without blocking the main thread.
- Managed using callbacks, promises, async/await.

Difference between `undefined`, `null`, and `NaN`

Answer:

- `undefined`: Variable declared but not assigned.
- `null`: Intentional absence of value.
- `NaN`: "Not a Number", result of invalid numeric operations.