

# What is Inheritance?

Inheritance allows one class to **use the properties and methods of another class**.

- The base class is called **Parent class / Super class**
- The derived class is called **Child class / Subclass**

It helps in:

- Reusability
- Reducing code duplication
- Creating hierarchical relationships

## Syntax of Inheritance in JavaScript

Use the `extends` keyword.

### Example

```
class Animal {  
    eat() {  
        console.log("Animal eats food");  
    }  
}  
  
class Dog extends Animal {  
    bark() {  
        console.log("Dog barks");  
    }  
}  
  
let d = new Dog();  
d.eat(); // from parent class  
d.bark(); // from child class
```

### Constructor in Inheritance

If both classes have constructors, then child class must use `super()` inside the constructor.

### Example

```
class Parent {  
    constructor(name) {  
        this.name = name;  
    }  
}  
  
class Child extends Parent {  
    constructor(name, age) {  
        super(name); // calling Parent constructor  
        this.age = age;  
    }  
}
```

```
let c = new Child("Neeraj", 23);
```

## What is super Keyword?

super is used to:

### A. Call parent class constructor

```
super(arguments)
```

### B. Access parent class methods

```
super.methodName()
```

## Using super() in Constructor

### Example

```
class Vehicle {  
    constructor(company) {  
        this.company = company;  
    }  
}  
  
class Car extends Vehicle {  
    constructor(company, model) {  
        super(company); // parent constructor  
        this.model = model;  
    }  
}  
  
let obj = new Car("Toyota", "Fortuner");  
console.log(obj.company);
```

## Method Overriding

When child class defines a **method with the same name** as the parent class method, it is called **method overriding**.

The child method replaces the parent method.

### Example

```
class A {  
    show() {  
        console.log("Parent show()");  
    }  
}  
  
class B extends A {  
    show() {  
        console.log("Child show()");  
    }  
}
```

```
let obj = new B();
obj.show(); // Child show()
```

## Using super to Call Parent Method During Override

### Example

```
class A {
  show() {
    console.log("Parent show()");
  }
}

class B extends A {
  show() {
    super.show(); // calling parent method
    console.log("Child show()");
  }
}

let obj = new B();
obj.show();
```

### Output:

```
Parent show()
Child show()
```

## Real-life Example of Inheritance + Overriding

```
class BankAccount {
  constructor(owner, balance) {
    this.owner = owner;
    this.balance = balance;
  }

  withdraw(amount) {
    this.balance -= amount;
    console.log("New balance:", this.balance);
  }
}

class SavingsAccount extends BankAccount {
  withdraw(amount) {
    if (amount > this.balance) {
      console.log("Insufficient funds");
      return;
    }
    super.withdraw(amount); // call parent logic
  }
}

let s = new SavingsAccount("Neeraj", 5000);
s.withdraw(6000); // child override
s.withdraw(2000); // parent method via super
```

## Summary Table

Topic	Meaning
extends	Creates child class
super()	Calls parent constructor
super.method()	Calls parent method
Method Overriding	Rewriting parent method in child
Inheritance	Reusing parent properties & methods

## JavaScript Notes – Getters, Setters, Static Methods & instanceof Operator

### Getters and Setters

Getters and setters are **special class methods** used to **get** and **set** values of object properties in a controlled way.

They help in:

- Validation
- Protecting internal data
- Making code look clean (like reading a normal property)

### Getter (get)

A **getter** returns the value of a property.

### Syntax

```
get propertyName() { ... }
```

### Example

```
class User {
  constructor(name) {
    this._name = name;
  }

  get name() {
    return this._name.toUpperCase();
  }
}

let u = new User("Neeraj");
console.log(u.name); // NEERAJ
```

`_name` is used as a **private-like property** (convention only).

### Setter (set)

A **setter** updates the value of a property with validation.

## Syntax

```
set propertyName(value) { ... }
```

## Example

```
class User {
  constructor(name) {
    this._name = name;
  }

  set name(value) {
    if (value.length < 3) {
      console.log("Name too short");
      return;
    }
    this._name = value;
  }
}

let u = new User("Ram");
u.name = "Jo"; // invalid → Name too short
u.name = "Aman"; // valid
```

## Static Methods

Static methods belong to the **class itself**, not to the objects.

They are used for:

- Utility functions
- Helper methods
- Common operations

## Syntax

```
static methodName() { ... }
```

## Example

```
class MathUtils {
  static add(a, b) {
    return a + b;
  }
}

console.log(MathUtils.add(5, 7)); // 12
```

You **cannot** call static methods on objects.

```
let m = new MathUtils();
m.add(4,5); // Error
```

## Static Properties

You can also create static properties.

```
class Counter {  
    static count = 0;  
  
    static increment() {  
        this.count++;  
    }  
}  
  
Counter.increment();  
console.log(Counter.count); // 1
```

## instanceof Operator

`instanceof` checks whether an object belongs to a particular class.

### Syntax

```
object instanceof ClassName
```

### Example

```
class Car {}  
let c = new Car();  
  
console.log(c instanceof Car);      // true  
console.log(c instanceof Object);  // true (all classes inherit Object)
```

### instanceof with Inheritance

```
class Animal {}  
class Dog extends Animal {}  
  
let d = new Dog();  
  
console.log(d instanceof Dog);      // true  
console.log(d instanceof Animal);  // true  
console.log(d instanceof Object);  // true
```

`instanceof` checks **the entire prototype chain**, not just direct parent.

### Full Example (Getters + Setters + Static + instanceof)

```
class Student {  
    static count = 0;  
  
    constructor(name, age) {  
        this._name = name;  
        this._age = age;  
        Student.count++; // accessing static property  
    }  
  
    get name() {  
        return this._name;  
    }  
}
```

```

set age(value) {
    if (value < 5) {
        console.log("Age too low");
        return;
    }
    this._age = value;
}

static totalStudents() {
    return Student.count;
}
}

let s1 = new Student("Neeraj", 23);
let s2 = new Student("Aman", 20);

console.log(Student.totalStudents()); // 2
console.log(s1 instanceof Student); // true
console.log(s1 instanceof Object); // true

```

## Summary Table

Feature	Meaning
<b>get</b>	Reads a property like a normal variable
<b>set</b>	Sets/validates a property
<b>static method</b>	Belongs to the class, not objects
<b>static property</b>	Shared value across the class
<b>instanceof</b>	Checks if object belongs to class

## Local Scope & Global Scope

### Global Scope

A variable that is declared **outside any function or block** is global.  
It can be accessed **anywhere** in the program.

### Example

```

let globalVar = "I am global";

function test() {
    console.log(globalVar); // Accessible inside the function
}

test();
console.log(globalVar); // Also accessible here

```

### Local Scope

A variable declared **inside a function or block {}** is local.  
It can be accessed **only inside that block/function**.

## Example

```
function demo() {  
    let localVar = "I am local";  
    console.log(localVar); // Accessible  
}  
  
demo();  
console.log(localVar); // ✗ Error (not accessible outside)
```

## Block Scope (with let/const)

`let` and `const` are **block-scoped**.

```
if (true) {  
    let x = 10;  
    const y = 20;  
}  
  
console.log(x); // ✗ Error  
console.log(y); // ✗ Error
```

## IIFE (Immediately Invoked Function Expression)

IIFE = A function that runs **immediately** after it is defined.  
Used to create **private scope** and avoid polluting the global scope.

### Syntax

```
(function() {  
    console.log("IIFE executed!");  
})();
```

### With Parameters

```
(function(name) {  
    console.log("Hello " + name);  
) ("Neeraj");
```

### Arrow Function IIFE

```
(( ) => {  
    console.log("Arrow IIFE");  
)();
```

## Function Hoisting

Hoisting = JavaScript moves **function declarations and var declarations** to the top of the scope before execution.

### Function Declarations ARE hoisted

You can call the function **before** it is defined.

```
greet(); // Works because hoisted

function greet() {
    console.log("Hello!");
}
```

## Function Expressions are NOT hoisted (let/const)

```
hello(); // Error

let hello = function() {
    console.log("Hi!");
};
```

## Arrow Functions are NOT hoisted (let/const)

```
say(); // Error

const say = () => {
    console.log("Arrow function");
};
```

## Variable Hoisting

### var is hoisted but initialized as undefined

```
console.log(a); // undefined (not error)
var a = 10;
```

### let/const are hoisted BUT NOT initialized

They stay in **Temporal Dead Zone (TDZ)** until line of declaration.

```
console.log(b); // Error
let b = 20;
```

## Compact Table

Concept	Meaning	Example
Global Scope	Accessible everywhere	let x = 10;
Local Scope	Accessible only in function	function f(){ let y=20 }
Block Scope	Accessible only inside {}	if(){ let z=30 }
IIFE	Auto-running function	(function(){}())()
Function Hoisting	Function declarations move up	greet() before declaring
Var Hoisting	Hoisted as undefined	console.log(a); var a=10
Let/Const Hoisting	TDZ until declared	console.log(b); let b=10

# Fetch API (Modern way to make HTTP requests)

The **Fetch API** is used to send requests (GET, POST, PUT, DELETE) to a server.  
It returns a **Promise**.

## Example: GET Request

```
fetch("https://api.example.com/data")
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.log("Error:", error));
```

## Example: POST Request

```
fetch("https://api.example.com/users", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({
    name: "Neeraj",
    age: 20
  })
})
.then(res => res.json())
.then(data => console.log(data))
.catch(err => console.log(err));
```

# Local Storage (Stored permanently until cleared)

- Stores data **without expiration date**
- Data stays even after closing the browser
- Max storage ~10MB
- Stores only **strings**

## Save Data

```
localStorage.setItem("name", "Neeraj");
```

## Get Data

```
let name = localStorage.getItem("name");
console.log(name);
```

## Remove Data

```
localStorage.removeItem("name");
```

## Clear All

```
localStorage.clear();
```

## Store Objects (JSON)

```
let user = { name: "Neeraj", age: 20 };

localStorage.setItem("user", JSON.stringify(user));
```

Retrieve:

```
let data = JSON.parse(localStorage.getItem("user"));
console.log(data.name);
```

## Session Storage (Clears when tab/browser closes)

- Similar to local storage
- Data is removed when **tab or browser closes**
- Good for **temporary data**, forms, cart items, session info

### Save Data

```
sessionStorage.setItem("token", "12345");
```

### Get Data

```
let t = sessionStorage.getItem("token");
console.log(t);
```

### Remove Data

```
sessionStorage.removeItem("token");
```

### Clear All

```
sessionStorage.clear();
```

## Cookies in JavaScript

- Small data stored in browser
- Max size around **4 KB**
- Can have **expiration time**
- Sent automatically with **HTTP requests**

### Set Cookie

```
document.cookie = "username=Neeraj; expires=Wed, 21 Aug 2025 12:00:00 UTC;
path=/";
```

### Read Cookies

```
console.log(document.cookie);
```

### Delete Cookie

```
document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 UTC;
path=/";
```

## Difference Between LocalStorage, SessionStorage & Cookies

Feature	LocalStorage	Session Storage	Cookies
Storage Size	~10MB	~5MB	~4KB
Expires	Never	Close tab/browser	Set by server
Sent with HTTP requests	No	No	Yes
Use Case	Long-term data	Temporary data	Login, tracking

### Short Summary

- **Fetch API** → For calling APIs
- **LocalStorage** → Permanent storage
- **Session Storage** → Temporary tab-based storage
- **Cookies** → Small data + backend usage

## Prototype in JavaScript

JavaScript uses **prototype-based inheritance**, not classical inheritance like Java or C++.

Every function in JavaScript has a property called **prototype**, which is an **object used for inheritance**.

### How it works

When you create an object using a constructor function:

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

Person.prototype.sayHello = function () {
  console.log("Hello " + this.name);
};

let p1 = new Person("Neeraj");
p1.sayHello(); // works!
```

### Why it works?

Because JavaScript looks for `sayHello` inside `p1`.  
If not found → it looks inside `Person.prototype`.

This is called **Prototype Chain**.

## 2. `__proto__` (Internal Prototype Link)

Every JavaScript object contains a hidden property called `[Prototype]`, also accessible as:

```
object.__proto__
```

`__proto__` is a **reference** to the prototype that an object inherits from.

### Example

```
let obj = {};
console.log(obj.__proto__);
```

- This shows all methods inherited from `Object.prototype`.
- `__proto__` connects the object to the prototype chain.

### Relationship between `prototype` & `__proto__`

Concept	Meaning
<code>prototype</code>	Exists only on <b>functions</b> ; used to create inheritance
<code>__proto__</code>	Exists on <b>objects</b> ; points to the prototype it inherits

### Diagram (Easy to remember)

```
function Person() {}    → has: Person.prototype
new Person() → object   → obj.__proto__ === Person.prototype
```

## JavaScript Closure

A **closure** is created when:

- A **function remembers** the variables from its **parent scope**,
- Even after the parent function has finished executing.

### Example

```
function outer() {
  let x = 10;

  function inner() {
    console.log(x); // inner remembers x
  }

  return inner;
}

let fn = outer();
fn(); // prints 10
```

Even though `outer()` has finished,  
`inner()` still has access to **x = 10** → this is a closure.

## Why Closures Are Useful?

1. **Data privacy / Encapsulation**
2. **Creating private variables**
3. **Function factories**
4. **Event listeners**
5. **Asynchronous code**

## Closure Example (Private Variable)

```
function counter() {  
  let count = 0;  
  
  return function () {  
    count++;  
    console.log(count);  
  };  
}  
  
let c = counter();  
c(); // 1  
c(); // 2  
c(); // 3
```

The variable `count` is **protected** inside the function.

## Regular Expressions in JavaScript (RegEx)

A **Regular Expression** is a pattern used to match, search, or replace text.

You create RegEx in two ways:

### 1. Literal Syntax

```
let regex = /hello/;
```

### 2. Constructor Syntax

```
let regex = new RegExp("hello");
```

## Common RegEx Methods

### 1. `test()` → returns true/false

```
/hello/.test("hello world"); // true
```

### 2. `match()` → returns matched results

```
"hello world".match(/world/); // ["world"]
```

### 3. `replace()`

```
"hello world".replace(/world/, "Neeraj");
// output: "hello Neeraj"
```

## RegEx Special Characters

### Character Classes

Pattern	Meaning
\d	Digit (0–9)
\w	Word char (a-z, A-Z, 0–9, _)
\s	Whitespace
\D	Non-digit
\W	Non-word char
\S	Non-whitespace

### Quantifiers

Pattern	Meaning
+	1 or more
*	0 or more
?	0 or 1
{n}	Exactly n times
{n, }	n or more
{n, m}	Between n and m

Example:

```
/\d+/.test("1234"); // true
```

### Anchors

Pattern	Meaning
^	Start of string
\$	End of string

Example:

```
/^hello/.test("hello world"); // true
/world$/.test("hello world"); // true
```

### Groups and Alternation

```
/(cat|dog)/.test("I have a dog"); // true
```

### Example: Validate Email

```
let emailRegex = /^[\\w.-]+@[\\w.-]+\\.\\.[a-zA-Z]{2,}\\$/;  
emailRegex.test("test@gmail.com"); // true
```

## Event Loop in JavaScript

JavaScript is **single-threaded**, meaning it executes **one task at a time**.

But it handles **asynchronous tasks** (setTimeout, promises, API calls) using the **Event Loop**.

### Event Loop Components

#### 1. Call Stack

- Executes synchronous code line by line.

#### 2. Web APIs

- Browser-managed async functions (setTimeout, fetch, DOM events)

#### 3. Callback Queue (Task Queue)

- Stores callbacks from Web APIs (like setTimeout, click events)

#### 4. Microtask Queue

- Stores Promise .then() callbacks
- Higher priority than callback queue

### How Event Loop Works?

JS runs synchronous code in **call stack**

Async task goes to **Web API**

After completion:

- **Promises go to Microtask Queue**
- **setTimeout / events go to Callback Queue**

**Event Loop** checks:

- If call stack is empty → push microtasks first
- After microtasks → push callback queue tasks

### Very Important: Microtasks > Macrotasks

**Microtasks (Promises) run first** even if setTimeout has zero delay.

## **Example:**

```
console.log("A");  
  
setTimeout(() => console.log("B"), 0);  
  
Promise.resolve().then(() => console.log("C"));  
  
console.log("D");
```

## **Output:**

```
A  
D  
C ← Promise (microtask)  
B ← setTimeout (callback queue)
```

## **Example Breakdown**

```
console.log("start");  
  
setTimeout(() => {  
  console.log("timeout");  
}, 0);  
  
Promise.resolve().then(() => {  
  console.log("promise");  
});  
  
console.log("end");
```

## **Output:**

```
start  
end  
promise ← microtask  
timeout ← macrotask
```

## **Summary (Important for Interview & Exams)**

### **Regular Expressions**

- Used for pattern matching.
- Supports character classes, quantifiers, anchors, groups.
- Common methods: `test`, `match`, `replace`.

### **Event Loop**

- JS is single-threaded.
- Event Loop handles async operations.
- Microtask Queue > Callback Queue.

Promises **run** before `setTimeout`.