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

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

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
get	Reads a property like a normal variable
set	Sets/validates a property
static method	Belongs to the class, not objects
static property	Shared value across the class
instanceof	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	Local Storage	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
- **Local Storage** → Permanent storage
- **Session Storage** → Temporary tab-based storage
- **Cookies** → Small data + backend usage