CLASSES & Objects

CREATED BY :

PROTOTYPES in JavaScript

- a javascript object is an entity having **state** and **behavior** (properties and method).
- JS objects have a special property called prototype.
- We can set prototype using: `__proto__`.

⚠ If object & prototype have same method, object's method will be used.

Js JavaScript, there are different ways to create an Object

Top is one of the way to create an Object.

```
const student = {
}
```

Top is the easy way to create an Object

Little ones we had created one Object in Js but imagine we will have the another way to create an Object.

```
const student = {
  name: "Subhendu Thakur",
  age: 24,
}

const student2 = {
  name: "Subhendu Thakur",
  age: 24,
}
```

Prototype / State

Twice we use prototype of our Object.

Note: But we can also add/create new methods inside the Object.

You can write methods in 3 ways :

1. OBJECT LITERAL METHOD SYNTAX

Top is the both common and modern way

```
{
  fullname: "Subhendu Thakur",
  age: 24,
  printAge() {
    console.log("age = ", this.age);
  },
}
```

Method →

2. FUNCTION EXPRESSION AS A PROPERTY

Here pick single + function is a key.

```
{
  fullname: "Subhendu Thakur",
  age: 24,
  printAge: function() {
    console.log("age = ", this.age);
  },
}
```

3. ARROW FUNCTION AS A METHOD

⚠ This is not recommended for methods that use this, because **arrow functions** don't have their own **this**.

Every Object whatever is created inside of JavaScript , is Virtual Property automatically gets created.

Which is Prototype

```
{fullname: "Subhendu Thakur", age: 24, printAge: f} ⓘ
  age: 24
  fullname: "Subhendu Thakur"
  printAge: f ()
  ▶ [[Prototype]]: Object
```

The Prototype is an Object in itself

```
{fullname: "Subhendu Thakur", age: 24, printAge: f} ⓘ
  age: 24
  fullname: "Subhendu Thakur"
  printAge: f ()
  ▶ [[Prototype]]: Object
    ▶ constructor: f Object()
    ▶ hasOwnProperty: f hasOwnProperty()
    ▶ isPrototypeOf: f isPrototypeOf()
    ▶ propertyIsEnumerable: f propertyIsEnumerable()
    ▶ toLocaleString: f toLocaleString()
    ▶ toString: f toString()
    ▶ valueOf: f valueOf()
    ▶ __defineGetter__: f __defineGetter__()
    ▶ __defineSetter__: f __defineSetter__()
    ▶ __lookupGetter__: f __lookupGetter__()
    ▶ __lookupSetter__: f __lookupSetter__()
    ▶ __proto__: null
    ▶ get __proto__: f __proto__()
    ▶ set __proto__: f __proto__()
```

And inside of the prototype we can see different methods & different properties.

What is a Prototype in JavaScript?

Every object in JavaScript has a hidden internal property called [[Prototype]]. (you can access it via `__proto__`). This **prototype** acts like a blueprint — it's a property or method isn't found on the object itself. JavaScript looks in the prototype.

When you define methods inside a class, they don't go **inside the object**, but rather they live on the **prototype** of the class.

🔍 What's Happening Here?

You are **manually setting a custom prototype** for the sam object.

🧠 1. What is `__proto__` ?

- `__proto__` is a special internal property of every object.
- **It points to another object**, which is used as a **fallback for properties and methods** — this is how **prototype chaining** works:

```
const employee = {
  calcTax() {
    console.log("Tax rate is 10%");
  }
}

const sam = {
  salary: "1400$"
}

Object.setPrototypeOf(sam, employee);
```

3. When You Set `sam.__proto__ = employee;`

You're telling JavaScript:

"Any JS object, if you don't find a property in sam, look in employee next."

This is called **Prototype Inheritance**.

Now sam will **inherit** the method `calcTax()` from **employee**.

Default Object Structure of `employee` :

```
const employee = {
  calcTax() {
    console.log("Tax rate is 10%");
  }
}
```

```
▶ [[prototype]]: Object
  ▶ calcTax: f calcTax()
  ▶ [[Prototype]]: Object
```

2. Default Object Structure Before `__proto__` is set :

```
const sam = {
  salary: "1400$"
}
```

```
▶ {salary: "1400$"} ⓘ
  salary: "1400$"
  ▶ [[Prototype]]: Object
    ▶ constructor: f Object()
    ▶ hasOwnProperty: f hasOwnProperty()
    ▶ isPrototypeOf: f isPrototypeOf()
    ▶ propertyIsEnumerable: f propertyIsEnumerable()
    ▶ toLocaleString: f toLocaleString()
    ▶ toString: f toString()
    ▶ valueOf: f valueOf()
    ▶ __defineGetter__: f __defineGetter__()
    ▶ __defineSetter__: f __defineSetter__()
    ▶ __lookupGetter__: f __lookupGetter__()
    ▶ __lookupSetter__: f __lookupSetter__()
    ▶ __proto__: null
    ▶ get __proto__: f __proto__()
    ▶ set __proto__: f __proto__()
```

This object has no `calcTax()` method on its own.

What if the **employee** doesn't have the `calcTax()` method?

What would be the prototype chain look like?

`sam → employee → Object.prototype → null`

Where is sam's default prototype ?

You want to know :

Is I set **`employee` in sam.__proto__ where did sam's original/default** `prototype` **go? Where are the default methods** like `toString()`, `hasOwnProperty()` etc.?

🚀 The Truth:

When you **manually override** the prototype like this:

`sam.__proto__ = employee;`

You are **replacing** the default prototype that sam would have gotten from `Object.prototype`.

🚀 By default :

```
const sam = {
  salary: "1400$"
}
```

```
▶ {salary: "1400$"} ⓘ
  salary: "1400$"
  ▶ [[Prototype]]: Object
    ▶ constructor: f Object()
    ▶ hasOwnProperty: f hasOwnProperty()
    ▶ isPrototypeOf: f isPrototypeOf()
    ▶ propertyIsEnumerable: f propertyIsEnumerable()
    ▶ toLocaleString: f toLocaleString()
    ▶ toString: f toString()
    ▶ valueOf: f valueOf()
    ▶ __defineGetter__: f __defineGetter__()
    ▶ __defineSetter__: f __defineSetter__()
    ▶ __lookupGetter__: f __lookupGetter__()
    ▶ __lookupSetter__: f __lookupSetter__()
    ▶ __proto__: null
    ▶ get __proto__: f __proto__()
    ▶ set __proto__: f __proto__()
```

JavaScript sets `sam.__proto__ = Object.prototype` internally.

That's how sam gets access to default methods like:

- `hasOwnProperty()`
- `toString()`
- `valueOf()`

❗ But when you do this :

`sam.__proto__ = employee;`

You are **removing** the default prototype (`Object.prototype`) from sam and setting **employee as the prototype** of sam.

Now :

`sam + employee + Object.prototype`

So sam can **still access all the default methods**, but not directly.
It goes **through employee** first.

⚠ **Note:**
Setting `__proto__` directly is **not recommended in production code**.
It's a very old feature deprecated in some contexts.
Instead, use:

`Object.setPrototypeOf(sam, employee);`

What happens, if we have same name methods?

```
const employee = {
  calcTax() {
    console.log("Tax rate is 10%");
  }
}

const sam = {
  calcTax() {
    console.log("Tax rate is 20%");
  },
  salary: "1400$",
}

Object.setPrototypeOf(sam, employee);
```

```
▶ {salary: "1400$", calcTax: f} ⓘ
  ▶ calcTax: f calcTax()
  salary: "1400$"
  ▶ [[Prototype]]: Object
    ▶ calcTax: f calcTax()
    ▶ [[Prototype]]: Object
```

```
calcTax() {
  console.log("Tax rate is 20%");
},
```

```
calcTax() {
  console.log("Tax rate is 10%");
}
```

You'll get this, but main question is when you call **`sam.calcTax()`** which one will run?

```
sam.calcTax()
Tax rate is 20%
```

- JavaScript first looks for `calcTax()` **inside the sam object** itself.
- It **finds it there** → ✅ So it **executes that version**, and **stops searching**.
- The method inside the employee prototype is **ignored** because JavaScript already found the method at the first level.

- If both the object (**`sam`**) and its prototype (**`employee`**) have a method with the **same name**, JavaScript always calls the **method defined on the object** itself first.
- This is called **method shadowing** — the own property **shadows** the one from the prototype.
- The method in the prototype is **not lost** — it's just **not used unless the object doesn't have its own method**.

Prototype Chain Here:

```
sam {
  calcTax: function() { "Tax rate is 20%" },
  salary: "1400$"
}
```

↓

Prototype

```
employee {
  calcTax: function() { "Tax rate is 10%" }
}
```

↓

Object.prototype

↓

null

Prototype Chaining :

Prototype chaining is a mechanism in JavaScript where objects are linked to other objects through their `[[Prototype]]`.

When you access a property or method on an object, JavaScript first looks for it on the object itself.
If it's not found, it continues searching up the prototype chain until it finds it or reaches `null`.

CLASSES in JavaScript

- Class is a **program-code template** for creating objects.
- Those objects will have some state (variables) & some behavior (functions) inside it.

```
class myClass {
  constructor() {
    // code here
  }
  myMethod() {
    // code here
  }
}

let obj = new myClass();
```

🔷 What is a Class in JavaScript?

- A **class** is a blueprint for creating objects with **shared structure and behavior**.

Before classes, we used **constructor functions**. But ES6 introduced the class syntax to make OOP (**Object-Oriented Programming**) in JavaScript cleaner and more intuitive.

Thing	What it really is
`class`	Special function (blueprint)
`constructor()`	Runs automatically when object is created
`this.property`	Assigns variables (values) to the object
`method()`	Function saved in the prototype

```
class Customer {
  state() {
    console.log("state");
  }

  city() {
    console.log("city");
  }

  addressCount() {
    this.incrementAddress(this);
  }
}
```

```
▶ Customer {}
  ▶ [[Prototype]]: Object
    ▶ constructor: class Customer
    ▶ addressCount: f addressCount()
    ▶ city: f city()
    ▶ state: f state()
    ▶ [[Prototype]]: Object
```

```
▶ {name: "John", age: 30} ⓘ
  name: "John"
  age: 30
  ▶ [[Prototype]]: Object
    ▶ constructor: class Customer
    ▶ addressCount: f addressCount()
    ▶ city: f city()
    ▶ state: f state()
    ▶ [[Prototype]]: Object
```

Knowledge

What is an Instance?

An **Instance** is an object created using a **class** (or constructor function) with the **`new`** keyword.

🔍 Is a class like a function?

✅ **Yes! In JavaScript, a class is actually a special type of function.**

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

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

When you do:

`console.log(typeof Person);`

You're saying :

"Hey class, give me a new object following your blueprint."

Under the hood, JavaScript secretly **converts** it into something like this:

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

Person.prototype.greet = function () {
  console.log("Hello, I'm", this.name);
};
```

😎 **Surprise!** a class is **syntactic sugar** for constructor functions + prototype methods.

🔍 Class is like a blueprint

You can think of a class as:

- A **blueprint** or **template** to create objects
- It bundles together:
  - **Properties** (like `name`, `age`)
  - **Methods** (like `greet()`)

Difference between Object Literal & Class

Using an Object Literal:

```
const user = {
  name: "John",
  greet() {
    console.log("Hello, I'm", this.name);
  },
};

console.log(user);  // You'll see both 'name' and 'greet' directly
```

Here, the `greet()` method is directly in the object — self on a literal prototype.

```
▶ {name: "John", greet: f} ⓘ
  ▶ greet: f greet()
    name: "John"
  ▶ [[Prototype]]: Object
```

Using a Class:

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

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

const u1 = new Person("John");

console.log(u1);  // Just shows the object (with only 'name')
```

🧠 `greet()` is not directly inside `u1`, but in `Person.prototype`

- That's why it's **memory-efficient** — all instances share this one method.

```
▶ Person {name: "John"} ⓘ
  name: "John"
  ▶ [[Prototype]]: Object
    ▶ constructor: class Person
    ▶ greet: f greet()
    ▶ [[Prototype]]: Object
```

Feature	Class	Object Literal
Method Location	On the prototype	Inside the object itself
Memory Usage	Efficient (shared methods)	Can be heavier if many objects are created
Inheritance	Easily supported	Not built-in

1 CONSTRUCTOR

```
class myClass {
  constructor() { // code here
  }
  myMethod() { // code here
  }
}
```

`constructor()` method is :

- automatically invoked by **`new`**
- initializes object

🔷 What is a constructor?

- The constructor is a **special method** in a class that is **automatically called** when you create a new object using the **`new`** keyword.

🔑 Key Points

Feature	Description
Special Method	Only one `constructor` is allowed per class.
Automatically Called	Runs as soon as the object is created with `new`.
Initializes Object	Sets up the values of instance properties.
this Keyword	Refers to the new object being created.
Optional Parameters	You can pass values when creating an object.

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

  greet() {
    console.log("Hi, my name is ${this.name} and I'm ${this.age} years old.");
  }
}

const john = new Person("John", 30);
john.greet();  // Output: Hi, my name is John and I'm 30 years old.
```

🧠 Breakdown

✅ **`class Person {}`**

- Defines a new class called Person.

✅ **`constructor()`**

- A **special method** for initializing new objects created from the class. It runs **automatically** when you use **`new`**.

✅ **`this`** keyword

- Refers to the current instance (e.g., John).

✅ **Method `greet()`**

- A regular function defined **inside the class** — it will be shared via the **prototype** (memory-efficient).

Knowledge

A function becomes a **constructor function** when it is:

1. Designed to initialize a new object (use `this`)
2. Called with the **`new`** keyword

🧠 First: What is a constructor really?

✅ In simple words:

A **constructor** is a function that runs **automatically** when you create an object using a **`class`**.
Its main job is to assign **values** to properties (yes, you're correct) 🎯.

🧪 Let's compare two ways of creating an object:

1. ✅ Using object literal:

```
const obj = {
  name: "Sam",
  age: 25
};
```

- Here, you are **manually assigning** values.
- No constructor is involved — this is a plain object literal.
- You are just saying, "Hey JavaScript, here's an object, with these values."

2. ✅ Using a class (this uses a constructor behind the scenes):

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

const obj = new Person("Sam", 25);
```

- Here, JavaScript creates an empty object (`{}`) for you first.
- Then it **automatically calls the constructor()**.
- Inside the constructor, it runs:

```
this.name = name;
this.age = age;
```

So the object becomes:

```
{
  name: "Sam",
  age: 25
}
```

So yes:

The constructor's main job is **assigning values** to object properties when the object is being created using the `new` keyword.

If we need to perform some setup or assign values when an object is created for the first time, we use a constructor. That setup work is done automatically by the constructor.

Feature	Object Literal	Constructor Function	Class Syntax (ES6)
✅ Syntax	`{ key: value }`	`function Func() { this.key = value; }`	`class MyClass { constructor() { this.key = ; } }`
✅ Uses `new` keyword	❌ No	✅ Yes	✅ Yes
✅ Constructor auto-called	❌ No constructor involved	✅ Yes (when using `new`)	✅ Yes (when using `new`)
✅ Where methods are stored	Inside the object itself	On function prototype	On class prototype
✅ Assigning values	Manually	Inside constructor using `this.key = value`	Inside constructor or class fields
✅ Can reuse for multiple objs	❌ Yes (must repeat keys)	✅ Yes (using `new`)	✅ Yes (using `new`)
✅ Prototype linkage	`obj.__proto__ → Object.prototype`	`obj.__proto__ → Func.prototype → Object.prototype`	`obj.__proto__ → Class.prototype → Object.prototype`
✅ Example	`const obj = {name: "Sam"}`	`function Emp() { this.name = "Sam"; }`	`class Emp { constructor() { this.name = "Sam"; } }`
✅ Ideal for	Small one-time objects	Before ES6 class system	Modern, clean, reusable OOP

Before ES6 introduced the class syntax in 2015, JavaScript developers used **constructor functions** to create reusable object blueprints.

📦 Before ES6: Constructor Functions

✅ **Example:**

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

Person.prototype.greet = function () {
  console.log("Hello, I'm", this.name);
};

const p1 = new Person("Sam", 26);
```

🧠 What's Happening:

1. `new Person("Sam", 26);`
   - Creates a new object
   - Sets `this.name` and `this.age`
   - Links the object to `Person.prototype`
2. `greet()` is not copied to every object — it's stored once on `Person.prototype`, just like in ES6 classes.

✅ After ES6: Classes Made It Cleaner

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

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

const p1 = new Person("Sam", 26);
```

Output and behavior are the same as the constructor function.

🧠 Final Line:

Before ES6, constructor functions were the standard way to create reusable object blueprints in JavaScript.
✅ ES6 introduced classes to make the same concept cleaner, easier, and more readable — but under the hood, both use **prototypes**.

Important

- **Class fields run first**
- **Constructor runs second**
- If both set the same property, **constructor wins**
- JavaScript does **not** merge them into one constructor — it executes them **in order**

Syntax	Context	Official Term
`name = "sam"`	Object literal	**Object property**
`name = "sam"`	Inside class	**Class field**
`this.name = "sam"`	Inside constructor	**Property assignment** (at runtime)

When you create an object using a **class** (or constructor function) with the **`new`** keyword — that's when the constructor is **automatically called**, and its job is to **assign values** to the new object using **`this.property = value`**

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

const p1 = new Person("Sam", 30);  // constructor auto-called
```

✅ The constructor is **automatically invoked** when you write **`new Person()`**, and it assigns **`name`** and **`age`**.

Summary

When you create an object with a class, the constructor is automatically invoked.
Otherwise, in direct object creation, constructor stays in the prototype but is not invoked.
Thus if find constructor in `obj.__proto__`, that it's not doing any work unless you call **`new Object()`** explicitly.

When you create an object using **`class`** or constructor function, it's **`new`** keyword — that's when the constructor is **automatically called**, and its job is to **assign values** to the new object using **`this.property = value`**

- No constructor is invoked.
- No automatic method runs.
- You manually define properties and values.

⭐ **No class, no new, no constructor call.**

Yet, `obj.__proto__` still points to **`Object.prototype`**, and you can see:

`console.log(obj.constructor);  // function Object() { [native code] }`

✅ But that constructor (`Object()`) is **not invoked by you**, nor is it responsible for assigning those values.

2 INHERITANCE

- Inheritance is passing down properties & methods from **parent class** to **child class**.

```
class parent {
}

class child extends parent {
}
```

What is Inheritance?

Inheritance means one class or object can **acquire the properties and methods** of another.
It allows **code reuse**, helps avoid duplication, and enables **hierarchies** (like parent → child relationships).

⚠ If Child & Parent have same method, child's method will be used. (Method Overriding)

🧬 JavaScript Inheritance Types

1. Prototypal Inheritance (used in functions & objects)

Objects inherit directly from other objects via prototype chains.

2. Class Inheritance (ES6+) ✅ Modern way:

Classes use the **`extends`** keyword to inherit from other classes.

We'll focus on **class-based inheritance**, which is the modern and most used pattern today.

🔧 Basic Inheritance Syntax

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

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

class Child extends Parent {
  constructor(name, age) {
    super(name);  // call Parent's constructor
    this.age = age;
  }

  showAge() {
    console.log("I'm ${this.age} years old");
  }
}

const kid = new Child("Sam", 24);
kid.greet();  // Hello, I'm Sam (inherited)
kid.showAge();  // I'm 24
```

🔍 Explanation

Concept	Meaning
`extends`	Makes Child inherit from Parent
`super(name)`	Calls the parent constructor and passes `name`
`this.age = ...`	Adds new properties in the child
Inherited methods	`greet()` is accessible on Child via prototype

⚠ Important Rules

✅ You MUST call `super()` in the child constructor before using `this`.

```
class Child extends Parent {
  constructor() {
    this.name = "John";  // ❌ Error: ReferenceError: Must call super() before accessing 'this'
    super();
  }
}
```

```
class Child extends Parent {
  name: greet() {  // ❌ ERROR: SyntaxError: Unexpected token ':'
  greet() {
    console.log("Hi, I'm Child");
  }
}
```

You **cannot use** `super` **directly inside the class body.**

`super` is only valid **inside methods, constructors,** or **getters/setters.**

🔁 Method Overriding

You can redefine a parent method in the child:

```
class Parent {
  greet() {
    console.log("Hi, I'm Parent");
  }
}

class Child extends Parent {
  greet() {
    console.log("Hi, I'm Child");  // overrides parent's method
  }
}

const c = new Child();
c.greet();  // Hi, I'm Child
```

Whenever we have a method with the **same name** inside both the **Parent class** and the **Child class**, the **Child class method runs first** — this is called **method overriding**.

This happens because of **JavaScript's prototype chaining**:

- When we call a method on an object, the **JavaScript engine first checks the object's prototype** (`Child.prototype`).
- If it **finds the method** there, it **executes** it immediately.
- If the method is **not found**, it moves up to the **Parent class's prototype**.
- If still not found, it continues to `Object.prototype`, and finally reaches `null`.

🚀 Summary for Notes :

Keyword	Role
`extends`	Sets up inheritance between classes
`super()`	Calls the parent constructor
`this`	Refers to the current instance
Method Overriding	Child can define the same method to replace parent's method
Prototype Chain	Used behind the scenes to connect parent & child

Inheritance Chain

`kid → Child.prototype → Parent.prototype → Object.prototype → null`

- JavaScript looks through this chain when calling any method like `kid.greet()`

3 STATIC

What is a Static Method?

A **static method** is a method that **belongs to the class itself**, not to the instances created from the class.

```
class print {
  first() {
    console.log("first");
  }

  static second() {
    console.log("second");
  }

  third() {
    console.log("third");
  }
}

const check = new print();
check.first();  // first ✅
check.second();  // TypeError: a second is not a function ❌
```

Static method is not available on instance — Why?

Because :

`second()` is a **static method**, and it is **not copied** to the instance or its prototype chain.

Only **non-static (regular)** methods are placed on the class's prototype.

```
console.log(check);
```

```
▶ print {} ⓘ
  ▶ [[Prototype]]: Object
    ▶ constructor: class print
    ▶ first: f first()
    ▶ third: f third()
    ▶ [[Prototype]]: Object
```

Static methods can be inherited from one class to another, but they **cannot be accessed through instances.**

```
class childPrint extends print {}

childPrint.second();  // second
```

- `childPrint` inherits everything from `print` (static also)

📦 Final Summary :

Term	Meaning
Static method	A method defined using the `static` keyword inside a class
Belongs to	The class itself, **not** its instances
Call on	The class, like `ClassName.method()`
Use cases	Utility functions, helpers, factory methods
Not available on	Objects created using `new`

4 GETTER / SETTER

Getter : special method that makes a property **readable**.

Setter : special method that makes a property **writeable**.

Validate and modify value when reading/writing a property.

These are called **Instance Properties**

```
class Person {
  constructor(firstName, lastName, age) {
    this._firstName = firstName;
    this._lastName = lastName;
    this._age = age;
  }

  set firstName(newFirstName) {
    if (typeof newFirstName === "string" && newFirstName.length > 0) {
      this._firstName = newFirstName;
    } else {
      console.error("First name must be a non-empty string");
    }
  }

  set lastName(newLastName) {
    if (typeof newLastName === "string" && newLastName.length > 0) {
      this._lastName = newLastName;
    } else {
      console.error("Last name must be a non-empty string");
    }
  }

  set age(newAge) {
    if (typeof newAge === "number" && newAge > 0) {
      this._age = newAge;
    } else {
      console.error("Age must be a non-negative number");
    }
  }

  get firstName() {
    return `First Name : ${this._firstName}`;
  }

  get lastName() {
    return `Last Name : ${this._lastName}`;
  }

  get fullName() {
    return `Full Name : ${this._firstName} ${this._lastName}`;
  }

  get age() {
    return `Age : ${this._age}`
  }
}

const person = new Person("Subhendu", "Thakur", 24);
console.log(person.firstName);
console.log(person.lastName);
console.log(person.fullName);

console.log(person.age);
```

🔷 What are Getters and Setters?

Getters (`get`) and **Setters** (`set`) are special methods in JavaScript classes used to **access** and control **instance properties** using **clean, property-like syntax**, while running custom logic behind the scenes.

✅ Purpose of Getter & Setter :

We use **getters** and **setters** to **access** (`get`) or **update** (`set`) the **instance properties** of an object in a controlled way.

Getters and **Setters** allow us to **read** or **write** to **instance properties**, while adding logic like formatting, validation, or access control.

📦 Final Summary :

Getter (`get`) is used to **read** a property with logic.
Setter (`set`) is used to **assign** a property with control or validation.
Both allow your object properties to look clean and behave smart, like "invisible functions" that trigger behind the scenes.