JavaScript OOP: Prototype, Classes & Inheritance

[[Prototype]] & Prototypal Inheritance

Concept:

- Every JavaScript object has an internal property `[[Prototype]]`.
- If a property is not found in the object, JavaScript looks up its prototype chain — **this is prototypal inheritance**.

```
let a = {
  name2: "Harry",
  language: "JavaScript",
  run: () => {
    alert("self run");
  }
};

let p = {
  run: () => {
    alert("run");
  }
};

p.__proto__ = {
  name: "Jackie"
};

a.__proto__ = p;
a.run();               // Alerts: "self run"
console.log(a.name);   // Outputs: Jackie (inherited from prototype)
```

Key Points:

- First priority is always the object itself.
- If not found, JavaScript climbs up the prototype chain.

Classes in JavaScript

Syntax:

```
class ClassName {
  constructor() {
    // initialize properties
  }
```

```
  methodName() {
    // define methods
  }
}
```

Example: Railway Booking Form

Version 1: Manual fill

```
class RailwayForm {
  submit() {
    alert(this.name + ": Your form is submitted for train number: " +
this.trainno);
  }
  cancel() {
    alert(this.name + ": This form is cancelled for train number: " +
this.trainno);
  }
  fill(givenname, trainno) {
    this.name = givenname;
    this.trainno = trainno;
  }
}

let harryForm = new RailwayForm();
harryForm.fill("Harry", 145316);
harryForm.submit();  // Harry: Your form is submitted...

let rohanForm1 = new RailwayForm();
rohanForm1.fill("Rohan", 222420);
rohanForm1.submit(); // Rohan: Your form is submitted...
rohanForm1.cancel(); // Rohan: This form is cancelled...
```

Version 2: Using Constructor

```
class RailwayForm {
  constructor(givenname, trainno, address) {
    console.log(`🚧 Constructor called for ${givenname} 🚧`);
    this.name = givenname;
    this.trainno = trainno;
    this.address = address;
  }

  preview() {
    alert(`${this.name}: Your form is for Train No. ${this.trainno} 🚂. Address:
${this.address}`);
```

```
  }

  submit() {
    alert(`${this.name}: Form submitted 🎖️`);
  }

  cancel() {
    alert(`${this.name}: Form cancelled ❌`);
    this.trainno = 0;
  }
}

let harryForm = new RailwayForm("Harry", 13488, "420, Pacific Ocean, Bihar");
harryForm.preview();
harryForm.submit();
harryForm.cancel();
harryForm.preview(); // Address stays, train number becomes 0
```

🧬 Class Inheritance (`extends` keyword)

```
class Animal {
  constructor(name, color) {
    this.name = name;
    this.color = color;
  }
  run() {
    console.log(`${this.name} is running 🐾`);
  }
  shout() {
    console.log(`${this.name} is shouting 📢`);
  }
}

class Monkey extends Animal {
  eatBanana() {
    console.log(`${this.name} is eating banana 🍌`);
  }
  hide() {
    console.log(`${this.name} is hiding 🙈`);
  }
}

let bruno = new Animal("Bruno", "white");
let chimpu = new Monkey("Chimpu", "orange");

bruno.shout();     // Bruno is shouting
chimpu.eatBanana(); // Chimpu is eating banana
chimpu.hide();      // Chimpu is hiding
```

🔄 Method Overriding with `super`

```
class Employee {
  constructor(name) {
    console.log(`${name} - Employee constructor`);
    this.name = name;
  }
  login() {
    console.log(`${this.name} has logged in 🔑`);
  }
  logout() {
    console.log(`${this.name} has logged out 👋`);
  }
  requestLeaves(leaves) {
    console.log(`${this.name} requested ${leaves} leaves 📝 - Auto approved`);
  }
}

class Programmer extends Employee {
  constructor(name) {
    super(name); // Call parent constructor
    console.log("☑️ Custom constructor for Programmer");
  }

  requestCoffee(x) {
    console.log(`${this.name} requested ${x} coffees ☕`);
  }

  requestLeaves(leaves) {
    super.requestLeaves(leaves); // Call parent method
    console.log("Bonus: One extra leave granted 🎁");
  }
}

let dev = new Programmer("Harry");
dev.login();              // Harry has logged in
dev.requestLeaves(3);     // Harry requested 3 leaves + 1 extra
dev.requestCoffee(2);     // Harry requested 2 coffees
```

🧩 Static Methods

```
class Tool {
  static greet() {
    console.log("🛠️ Hello from static method!");
  }
}

Tool.greet();  // ☑️ Works
```

```
let obj = new Tool();
// obj.greet(); // ❌ Error: greet is not a function
```

🧠 Static Method Use Case:

- Used when logic belongs to the class as a whole, not a specific instance.

```
class Animal {
  constructor(name) {
    this.name = Animal.capitalize(name);
  }
  walk() {
    console.log(`${this.name} is walking 🚶`);
  }

  static capitalize(str) {
    return str.charAt(0).toUpperCase() + str.slice(1);
  }
}

let jack = new Animal("jack");
jack.walk(); // Jack is walking
// jack.capitalize() // ❌ Not accessible on instances
```

🎧 Getters and Setters

```
class Person {
  constructor(name) {
    this._name = name; // Use a backing property
  }

  get name() {
    return this._name + " 👋";
  }

  set name(newName) {
    this._name = newName;
  }

  fly() {
    console.log(`${this._name} is flying 🧠`);
  }
}

let p = new Person("Darshan");
console.log(p.name);  // Darshan 👋
p.name = "Vasani";
console.log(p.name);  // Vasani 👋
```

☑ Summary Table

Concept	Keyword/Feature	Example / Notes
Prototype	`__proto__`	Inherited fallback mechanism
Class Definition	`class`	ES6 syntax for OOP
Constructor	`constructor()`	Initializes new object
Inheritance	`extends`	`class B extends A`
Super Constructor	`super()`	Calls parent class constructor/method
Method Overriding	Same method name	Child class re-defines parent method
Static Methods	`static` keyword	Called on class, not instance
Getters & Setters	`get` / `set`	Control access to object properties

🧍 Object Literals & Methods

```
const user = {
  username: "hitesh",
  loginCount: 8,
  signedIn: true,

  getUserDetails: function () {
    console.log(this); // 🔍 Refers to the `user` object
  }
};

// Usage
// user.getUserDetails();
```

🏗 Constructor Function & `new` Keyword

```
function User(username, loginCount, isLoggedIn) {
  this.username = username;
  this.loginCount = loginCount;
  this.isLoggedIn = isLoggedIn;

  this.greeting = function () {
    console.log(`👋 Welcome ${this.username}`);
  };

  return this; // 👍 Implicit with `new`
```

```
}

const userOne = new User("hitesh", 12, true);
const userTwo = new User("ChaiAurCode", 11, false);

console.log(userOne.constructor); // 👉 Points to `User` function
```

🤔 **Why use `new`?**

- Creates a new empty object
- Sets the prototype
- Binds `this`
- Returns the object automatically

👄 **Without `new`, `this` refers to the global object (or `undefined` in strict mode)**.

🧬 Function Borrowing with `call`

```
function SetUsername(username) {
  this.username = username;
  console.log("📞 SetUsername called");
}

function createUser(username, email, password) {
  SetUsername.call(this, username); // 👈 Borrow constructor

  this.email = email;
  this.password = password;
}

const chai = new createUser("chai", "chai@fb.com", "123");
console.log(chai); // ❌ `undefined` — because no `new` used
```

🏫 Class with Getters and Setters

```
class User {
  constructor(email, password) {
    this._email = email;
    this._password = password;
  }

  get email() {
    return this._email.toUpperCase(); // 🔓 Accessor
  }

  set email(value) {
    this._email = value;
```

```
  }

  get password() {
    return `${this._password}hitesh`;
  }

  set password(value) {
    this._password = value;
  }
}

const hitesh = new User("h@hitesh.ai", "abc");
console.log(hitesh.email); // 👉 H@HITESH.AI
```

🧑‍🎓 Class Inheritance

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

  logMe() {
    console.log(`👤 USERNAME: ${this.username}`);
  }
}

class Teacher extends User {
  constructor(username, email, password) {
    super(username); // ⬆ Parent constructor
    this.email = email;
    this.password = password;
  }

  addCourse() {
    console.log(`📚 A new course was added by ${this.username}`);
  }
}

const chai = new Teacher("chai", "chai@teacher.com", "123");
chai.logMe();

const masalaChai = new User("masalaChai");
masalaChai.logMe();

console.log(chai instanceof User); // ☑ true
```

📏 Property Descriptors

```
const chai = {
  name: 'ginger chai',
  price: 250,
  isAvailable: true,

  orderChai: function () {
    console.log("❌ Chai not prepared");
  }
};

console.log(Object.getOwnPropertyDescriptor(chai, "name"));

Object.defineProperty(chai, "name", {
  enumerable: true,
  // writable: false,
});

for (let [key, value] of Object.entries(chai)) {
  if (typeof value !== "function") {
    console.log(`${key} : ${value}`); // 🍵 ginger chai details
  }
}
```

🔐 ES6 Class + Prototypes (Behind the Scenes)

```
function User(username, email, password) {
  this.username = username;
  this.email = email;
  this.password = password;
}

User.prototype.encryptPassword = function () {
  return `${this.password}abc`;
};

User.prototype.changeUsername = function () {
  return this.username.toUpperCase();
};

const tea = new User("tea", "tea@gmail.com", "123");
console.log(tea.encryptPassword()); // 🔐 123abc
console.log(tea.changeUsername());  // 👉 TEA
```

🪄 Object.create & Accessors

```
const User = {
  _email: 'h@hc.com',
  _password: 'abc',

  get email() {
    return this._email.toUpperCase();
  },

  set email(value) {
    this._email = value;
  }
};

const tea = Object.create(User);
console.log(tea.email); // 📧 H@HC.COM
```

🧠 Functions as Objects

```
function multipleBy5(num) {
  return num * 5;
}

multipleBy5.power = 2;

console.log(multipleBy5(5));        // 25
console.log(multipleBy5.power);     // 2
console.log(multipleBy5.prototype); // {}
```

📓 Prototypes for Method Sharing

```
function createUser(username, score) {
  this.username = username;
  this.score = score;
}

createUser.prototype.increment = function () {
  this.score++;
};

createUser.prototype.printMe = function () {
  console.log(`🎯 Score is ${this.score}`);
};

const chai = new createUser("chai", 25);
// const tea = createUser("tea", 250); // ❌ Forgot `new`
chai.printMe();
```

new Keyword Summary

```
function User(email, password) {
  this._email = email;
  this._password = password;

  Object.defineProperty(this, "email", {
    get: function () {
      return this._email.toUpperCase();
    },
    set: function (value) {
      this._email = value;
    }
  });

  Object.defineProperty(this, "password", {
    get: function () {
      return this._password.toUpperCase();
    },
    set: function (value) {
      this._password = value;
    }
  });
}

const chai = new User("chai@chai.com", "chai");
console.log(chai.email); // ⚖️ CHAI@CHAI.COM
```

Object & Array Prototype Inheritance

```
let myHeros = ["thor", "spiderman"];

let heroPower = {
  thor: "hammer",
  spiderman: "sling",

  getSpiderPower: function () {
    console.log(`🕸️ Spidy power is ${this.spiderman}`);
  }
};

// Prototype injections
Object.prototype.hitesh = function () {
  console.log("🧠 hitesh is present in all objects");
};
```

```
Array.prototype.heyHitesh = function () {
  console.log("👋 Hitesh says hello");
};

myHeros.hitesh();
myHeros.heyHitesh();
// heroPower.heyHitesh(); // ❌ Error
```

🧬 Prototypal Inheritance Chain

```
const User = {
  name: "chai",
  email: "chai@google.com"
};

const Teacher = {
  makeVideo: true
};

const TeachingSupport = {
  isAvailable: false
};

const TASupport = {
  makeAssignment: "JS assignment",
  fullTime: true,
  __proto__: TeachingSupport
};

// Modern syntax
Object.setPrototypeOf(TeachingSupport, Teacher);
Teacher.__proto__ = User;
```

🔡 Extending String Prototype

```
let anotherUsername = "ChaiAurCode     ";

String.prototype.trueLength = function () {
  console.log(`📏 Original: "${this}"`);
  console.log(`☑️ True length: ${this.trim().length}`);
};

anotherUsername.trueLength();
"hitesh".trueLength();
"iceTea".trueLength();
```

🔓 Static Methods

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

  logMe() {
    console.log(`🧑 Username: ${this.username}`);
  }

  static createId() {
    return "123";
  }
}

const hitesh = new User("hitesh");
// hitesh.createId(); // ❌ Static method not available on instance

class Teacher extends User {
  constructor(username, email) {
    super(username);
    this.email = email;
  }
}

const iphone = new Teacher("iphone", "i@phone.com");
console.log(Teacher.createId()); // ☑️ 123
```

👨‍💻 1. Object Literals & Methods

🧠 **Theory**:

In JavaScript, objects are collections of key-value pairs. You can define methods (functions inside objects) that can use the `this` keyword to refer to properties of the same object.

📜 **Analogy**:

Think of an object as a **profile card**. It has:

- Name (`username`)
- Info (`loginCount`)
- Actions (`getUserDetails`)

📌 Example:

```
const user = {
  username: "hitesh",
```

```
  loginCount: 8,
  signedIn: true,
  getUserDetails: function () {
    console.log(this); // Refers to user object
  }
};
```

🧵 2. Constructor Function & `new` Keyword

🧠 **Theory**:

Constructor functions allow you to create **multiple objects** with the same structure. When used with the `new` keyword, they:

1. Create an empty object
2. Assign properties to it using `this`
3. Return the object

🧵 **Analogy**:

Imagine `User` is a **blueprint (constructor)** for building user profiles. Each time you use `new`, you build a new user.

📌 Example:

```
function User(username, loginCount, isLoggedIn) {
  this.username = username;
  this.loginCount = loginCount;
  this.isLoggedIn = isLoggedIn;
}
```

📞 3. Function Borrowing using `call`

🧠 **Theory**:

The `call()` method lets one function **borrow** another's behavior by changing the value of `this`.

🧵 **Analogy**:

It's like using someone else's cooking recipe, but swapping your own ingredients (`this`).

📌 Example:

```
function SetUsername(username) {
  this.username = username;
```

```
}
SetUsername.call(this, username); // Borrow logic
```

🏠 4. Classes with Getters & Setters

🧠 **Theory**:

- **Getters** retrieve a property (like reading a file 📖).
- **Setters** update it (like writing to a file 📝).

📜 **Analogy**:

Think of a **locker**: You can open (get) or lock (set) its content, but only via defined access rules.

📌 Example:

```
get email() {
  return this._email.toUpperCase();
}
set email(value) {
  this._email = value;
}
```

🧑‍🏫 5. Inheritance (Extending Classes)

🧠 **Theory**:

Inheritance allows one class to gain the properties and methods of another. The `extends` keyword is used to inherit, and `super()` calls the parent constructor.

📜 **Analogy**:

A **Teacher** is a kind of **User**, just with extra powers (methods) like `addCourse`.

📏 6. Property Descriptors

🧠 **Theory**:

Using `Object.defineProperty`, we can control:

- `enumerable`: shows in loops?
- `writable`: can be changed?
- `configurable`: can be deleted/modified?

📜 **Analogy**:

Think of property descriptors like **file permissions** — read-only, hidden, locked, etc.

🧬 7. Prototype & Prototypal Inheritance

🧠 **Theory**:

JavaScript objects inherit from other objects. Functions also have a `prototype` object to share methods across instances.

📝 **Analogy**:

Imagine a **family tree** 🧬. If a child doesn't have a trait (method), they look up the chain (parent → grandparent).

📌 Example:

```
User.prototype.greet = function() {
  console.log("Hello");
};
```

🖌️ 8. Object.create()

🧠 **Theory**:

This method creates a new object and sets its prototype to an existing object.

📝 **Analogy**:

Like **cloning an existing object**, but you can add new behavior too!

🎛️ 9. Functions as First-Class Citizens

🧠 **Theory**:

Functions in JS are treated like objects. They can:

- Have properties
- Be passed around
- Be returned

📝 **Analogy**:

Functions are like **Swiss army knives** — versatile tools with many attachments.

🧮 10. Sharing Methods via Prototypes

🧠 Theory:

To avoid memory wastage, common methods are added to the prototype so that all instances can access them without duplication.

📝 Analogy:

Like sharing one **remote control** 📺 among many TVs instead of giving each TV its own.

🔓 11. The Power of `new`

🧠 Theory:

When you use `new` with a function:

1. A new object is created
2. `this` points to it
3. It inherits from the constructor's prototype

🦸 12. Extending Global Prototypes (CAUTION ⚠)

🧠 Theory:

You can add methods to global prototypes like `Array.prototype`, but it's **dangerous** — can conflict with other code.

📝 Analogy:

Like changing **everyone's shoes** 🩴 — maybe cool for you, chaos for others.

🧬 13. Prototype Chains (`__proto__` and `Object.setPrototypeOf`)

🧠 Theory:

You can manually link objects using `__proto__` or `Object.setPrototypeOf()` to create a chain of inheritance.

📝 Analogy:

Think of it as **adopting traits from another object**, like a backup plan if something is missing.

🔡 14. String Prototypes

🧠 Theory:

You can add custom methods to built-in prototypes like `String`, but again, use caution.

📝 Analogy:

Like giving every **string a magic method** 🪄 — powerful, but don't overuse.

🔓 15. Static Methods

🧠 **Theory**:

`static` methods belong to the class, not the instance. They're useful for utility methods.

🛋️ **Analogy**:

Imagine a **help desk** 🆘 you can call without needing to create a user.

🆕 JavaScript `new` Keyword — Full Behind-the-Scenes Guide

🧠 What does `new` actually do?

When you run:

```
const user1 = new User("hitesh", 5, true);
```

The `new` keyword does **four powerful things under the hood** 👇

☑️ Step-by-Step Behind the Scenes (🔍)

🔧 1. Creates a new empty object `{}`

```
const obj = {};
```

➡️ JavaScript creates a **brand-new object** for you. This will eventually be returned as the result of the constructor function.

🔗 2. Links the new object to a prototype

```
obj.__proto__ = ConstructorFunction.prototype;
```

➡️ It sets the internal `[[Prototype]]` (`__proto__`) of the new object to the `.prototype` property of the constructor function. Now `obj` can access all methods defined on `ConstructorFunction.prototype`.

This is how prototypal inheritance works.

---

3. Binds `this` inside the constructor to the new object

```
ConstructorFunction.call(obj, ...args);
```

The constructor function is executed, and `this` now refers to the newly created object (`obj`). Any `this.property = value` assignments attach data to the new object.

---

4. Returns the new object

If the constructor function does **not** return its own object explicitly, `new` will return the newly created object.

```
return obj;
```

If the constructor **explicitly** returns an object, that object is returned instead.

Full Example

```
function User(username, loginCount, isLoggedIn) {
  this.username = username;
  this.loginCount = loginCount;
  this.isLoggedIn = isLoggedIn;
}

const user1 = new User("hitesh", 12, true);
```

Behind the scenes:

```
// Step 1:
const tempObj = {};

// Step 2:
tempObj.__proto__ = User.prototype;

// Step 3:
User.call(tempObj, "hitesh", 12, true);

// Step 4:
return tempObj;
```

Analogy: "Building a Custom Toy"

Imagine you're in a factory building action figures:

1. **Step 1:** You get an empty mold (object `{}`).
2. **Step 2:** You connect it to the central blueprint (prototype).
3. **Step 3:** You fill in the toy's details like name and strength using `this`.
4. **Step 4:** You send the toy off as a finished product.

That's exactly what `new` does!

Special Note: What if constructor returns its own object?

```
function MyFunction() {
  this.name = "Darshan";
  return { name: "Custom Object" };
}

const obj = new MyFunction();
console.log(obj.name); // "Custom Object"
```

If you **explicitly return an object**, `new` gives you that object, not `this`.

Recap Table:

Step	Behind the Scenes	Explanation
1	`const obj = {}`	Create a new empty object
2	`obj.__proto__ = Constructor.prototype`	Set the prototype chain
3	`Constructor.call(obj, args...)`	Bind `this` to the object
4	`return obj`	Return the object (or custom one)

Bonus: Why Use `.prototype` with `new`?

```
function User(name) {
  this.name = name;
}
User.prototype.greet = function () {
  console.log(`Hi, I'm ${this.name}`);
};

const u = new User("Darshan");
u.greet(); // ☑ Inherited from prototype
```

🎓 **Prototype methods are shared** — saves memory! Each object created via `new` doesn't copy the method — it inherits it via the prototype chain.

❌ What if You Forget `new`?

```
const u = User("Darshan");
console.log(u); // undefined
```

🙈 Without `new`, `this` refers to the global object (or `undefined` in strict mode). So properties are attached to the global scope accidentally!

🌓 Always Use `new` with Constructors

To protect against mistakes, some devs add a guard:

```
function User(name) {
  if (!(this instanceof User)) {
    return new User(name);
  }
  this.name = name;
}
```

🧠 Keywords Summary

Keyword	Meaning
`this`	Refers to the newly created object inside constructor
`prototype`	Where shared methods live
`__proto__`	Internal link to the constructor's prototype
`new`	Orchestrates object creation steps
`instanceof`	Checks prototype inheritance

👨‍👩‍👧‍👦 Inheritance in JavaScript

JavaScript supports **prototypal inheritance**—objects can **inherit properties and methods from other objects** using a chain-like structure known as the **prototype chain**.

🧪 Your Code Breakdown

```
const User = {
    name: "chai",
    email: "chai@google.com"
}

const Teacher = {
    makeVideo: true
}

const TeachingSupport = {
    isAvailable: false
}

const TASupport = {
    makeAssignment: 'JS assignment',
    fullTime: true,
    __proto__: TeachingSupport
}

Teacher.__proto__ = User

Object.setPrototypeOf(TeachingSupport, Teacher)
```

🔍 What's Happening?

This code builds a **prototype chain** manually between objects. Let's visualize it:

```
TASupport --> TeachingSupport --> Teacher --> User
```

📌 So if you try to access a property on `TASupport`, JavaScript will:

1. Look inside `TASupport`
2. If not found, look inside `TeachingSupport`
3. If not found, go to `Teacher`
4. Then `User`
5. Finally up to `Object.prototype`

🧠 Real-World Analogy: Inheritance Chain

Imagine this like a **company hierarchy**:

- 👩 `User`: Basic employee
- 👨‍🏫 `Teacher`: Inherits from User → has video abilities
- 👩‍💼 `TeachingSupport`: Works under Teacher → availability
- 👨‍🎓 `TASupport`: Interns → gets assignments from TeachingSupport, and everything above it

So a TA intern 🥷 gets all the powers (methods/properties) from those above them in the chain.

⚙️ Modern Way (instead of `__proto__`)

```
Object.setPrototypeOf(childObject, parentObject);
```

☑ Safer and more readable than manually setting `__proto__`.

```
Object.setPrototypeOf(TeachingSupport, Teacher);
Object.setPrototypeOf(Teacher, User);
```

🛠️ Prototype Chain in Action

```
console.log(TASupport.makeAssignment);   // 'JS assignment' ☑
console.log(TASupport.isAvailable);      // false ☑
console.log(TASupport.makeVideo);        // true ☑ (from Teacher)
console.log(TASupport.name);             // 'chai' ☑ (from User)
```

All inherited step-by-step via prototype chain 🪜.

🧬 Prototype Chain Summary:

Object	Inherits From	Properties Available
TASupport	TeachingSupport	makeAssignment, fullTime, isAvailable, makeVideo, name, email
TeachingSupport	Teacher	isAvailable, makeVideo, name, email
Teacher	User	makeVideo, name, email
User	Object.prototype	name, email, plus JS default object methods

---

✨ Bonus Part: Custom Prototype Method

```
String.prototype.trueLength = function(){
    console.log(`${this}`);
    console.log(`True length is: ${this.trim().length}`);
}
```

What's Happening?

📌 You are **adding a custom method** to all strings in JavaScript via the **String.prototype**.

```
"iceTea".trueLength();
// Outputs: "iceTea"
// True length is: 6
```

Even `" chai "`.trueLength() will trim whitespace and count only visible characters.

⚠️ Caution: Avoid Polluting Prototypes in Production

While it's fun and powerful to extend built-in types, like `String.prototype`, avoid it in production unless you're writing a library. Why? Because it can **conflict** with other code or polyfills.

📌 Key Concepts to Remember

Concept	Explanation
`__proto__`	Internal link between objects
`.prototype`	Used for method sharing (functions/constructors)
`Object.setPrototypeOf`	Safe way to set inheritance
`prototype chain`	Path followed when accessing properties
`String.prototype`	Allows adding methods to all string instances

📄 Summary

🔗 JavaScript uses **prototypal inheritance**, not class-based (though `class` is syntactic sugar). 📦 Objects inherit from other objects via their prototype. ⚙️ You can use `Object.setPrototypeOf` or `__proto__` to link objects. 💡 You can also extend built-in prototypes like `String.prototype`.

📚 Suggested Exercises

1. Create a custom `Array.prototype.sum()` method.
2. Make a chain of objects (e.g., CEO → Manager → Employee) and log properties from top-down.
3. Practice using `Object.create()` to set prototypes.

🔍 Understanding ES6 Classes Behind the Scenes in JavaScript

✨ The Magic of ES6 `class`

```
class User {
    constructor(username, email, password){
        this.username = username;
        this.email = email;
        this.password = password
    }

    encryptPassword(){
        return `${this.password}abc`
    }

    changeUsername(){
        return `${this.username.toUpperCase()}`
    }
}
```

🧪 Usage:

```
const chai = new User("chai", "chai@gmail.com", "123")
console.log(chai.encryptPassword());  // Output: 123abc
console.log(chai.changeUsername());   // Output: CHAI
```

🧠 Analogy: **Factory + Prototype Toolbox**

Think of a `class` like a **factory** that produces multiple users 🧍‍♂️ 🧍‍♀️, each with:

- Their own **properties** (`username`, `email`, `password`)
- And access to shared **methods** (`encryptPassword`, `changeUsername`) from a **shared prototype toolbox** 🧰

🔬 What's Happening Behind the Scenes?

The ES6 `class` is actually **syntactic sugar** over the **prototype-based inheritance** system.

So this:

```
class User {
    constructor(username, email, password){
        this.username = username;
        this.email = email;
        this.password = password
    }

    encryptPassword(){
        return `${this.password}abc`
    }
```

```
    changeUsername(){
        return `${this.username.toUpperCase()}`
    }
}
```

Is **equivalent** to this:

```
function User(username, email, password){
    this.username = username;
    this.email = email;
    this.password = password;
}

User.prototype.encryptPassword = function(){
    return `${this.password}abc`;
}

User.prototype.changeUsername = function(){
    return `${this.username.toUpperCase()}`;
}
```

🛠️ You are **manually attaching** the methods to the `User.prototype`, which is exactly what the class does internally.

🧱 Breakdown: ES6 vs Traditional

ES6 `class` Syntax	**Traditional `function` Syntax**
Uses `class` keyword	Uses `function` constructor
Methods defined inside class body	Methods added to `Function.prototype`
Cleaner, OOP-like syntax	More verbose but identical behavior
Hidden internal prototype setup	You set prototype manually

🔁 Visual Diagram (Prototype Chain)

```
chai
 |
 └──> User.prototype
        ├── encryptPassword()
        └── changeUsername()
```

Same chain for the traditional version:

```
tea
 |
 └──> User.prototype
         ├── encryptPassword()
         └── changeUsername()
```

🧠 Both `chai` and `tea` are **instances** of `User`, and both **share** the same methods via the prototype.

🧠 Real-World Analogy: Class = Cookie Cutter 🍪

- `class User` → 🍪 Cookie Cutter (Mould)
- `chai`, `tea` → Cookies made using the cutter
- `prototype` → Common recipe shared by all cookies
- `this.username`, `this.email` → Custom flavors per cookie 🍫🍓

🔍 `new` Keyword — Behind the Curtain 🎭

```
const chai = new User("chai", "chai@gmail.com", "123");
```

This does 4 things:

1. 🆕 Creates an empty object: `{}`
2. 🔗 Sets the prototype: `chai.__proto__ = User.prototype`
3. 🧠 Calls the constructor: `User.call(chai, ...)`
4. 📦 Returns the new object

So, internally it behaves like:

```
let chai = {};
chai.__proto__ = User.prototype;
User.call(chai, "chai", "chai@gmail.com", "123");
```

✨ That's the magic of `new` in JavaScript.

🧾 Summary

Concept	Meaning
`class`	Cleaner syntax to define constructor + methods
`constructor()`	Method called automatically with `new`
`prototype`	Shared location where all instance methods are stored

Concept	Meaning
`new`	Creates instance and sets up prototype chain
`this`	Refers to the current instance being constructed

🔄 Recap: Why Use `class`?

☑ Clean, readable syntax ☑ Easy OOP mental model ☑ Still uses the same prototype mechanism ☑ No performance difference from function-based constructors

---

☑ Exercises

1. Convert this ES6 `class` into its behind-the-scenes version:

```
class Product {
    constructor(name, price){
        this.name = name;
        this.price = price;
    }

    discount(percent){
        return this.price * ((100 - percent)/100);
    }
}
```

2. Build a prototype chain: `Admin → Moderator → User`, and test method lookup.
3. Add a method to `Array.prototype` like `.average()` and use it on arrays.

🧬 Topic: Inheritance in JavaScript (ES6 Classes)

🧠 What is Inheritance?

Inheritance allows one class (child 👶) to use the **properties and methods** of another class (parent 👨‍👩‍👧), so you don't repeat code — you **extend it**.

In JavaScript, this is done using the `extends` keyword.

🔨 Your Code:

```
class User {
    constructor(username){
        this.username = username
    }
```

```
    logMe(){
        console.log(`USERNAME is ${this.username}`);
    }
}

class Teacher extends User {
    constructor(username, email, password){
        super(username); // calls User's constructor
        this.email = email;
        this.password = password;
    }

    addCourse(){
        console.log(`A new course was added by ${this.username}`);
    }
}

const chai = new Teacher("chai", "chai@teacher.com", "123")
chai.logMe(); // USERNAME is chai

const masalaChai = new User("masalaChai")
masalaChai.logMe(); // USERNAME is masalaChai

console.log(chai instanceof User); // true
```

Analogy: Inheritance as Family Traits

- `User`: Parent class — knows how to `logMe()`
- `Teacher`: Child class — inherits `logMe()` and adds `addCourse()`
- `super(username)` → like saying "Hey dad, here's your part of the family DNA"
- Both `chai` and `masalaChai` can use `logMe()` since it comes from the parent

---

What's happening behind the scenes?

Step-by-Step:

```
// Parent Class
function User(username){
    this.username = username;
}
User.prototype.logMe = function(){
    console.log(`USERNAME is ${this.username}`);
}

// Child Class
function Teacher(username, email, password){
    // Call the parent constructor
    User.call(this, username);
```

```
    this.email = email;
    this.password = password;
}

// Inherit prototype chain
Teacher.prototype = Object.create(User.prototype);
Teacher.prototype.constructor = Teacher;

// Add method specific to Teacher
Teacher.prototype.addCourse = function(){
    console.log(`A new course was added by ${this.username}`);
}

// Create instances
const chai = new Teacher("chai", "chai@teacher.com", "123");
chai.logMe();
chai.addCourse();

const masalaChai = new User("masalaChai");
masalaChai.logMe();
```

---

super() Keyword

- `super()` is used in child class to call **constructor of the parent class**.
- Without `super()`, you **cannot access** `this` in a child class constructor.

> It's like calling your parent to ask for inherited family traits before adding your own.

---

instanceof Keyword

```
console.log(chai instanceof User); // true
```

Checks if an object's prototype chain includes the constructor's prototype.

```
chai --> Teacher.prototype --> User.prototype --> Object.prototype
```

Since `User.prototype` is in the chain, it's `true`.

Full Inheritance Chain Diagram

```
chai (instance of Teacher)
   |
   └──> Teacher.prototype
```

```
                |
                └──> User.prototype
                            |
                            └──> Object.prototype
```

☑ Summary Table

Concept	Purpose	Example
`extends`	Inherit from another class	`class Teacher extends User`
`super()`	Call parent class constructor	`super(username)`
`instanceof`	Check prototype inheritance	`chai instanceof User`
Method Lookup	Looks in own class → parent class → Object	`chai.logMe()` works via User

---

🪄 Bonus: Try This

```
console.log(chai instanceof Teacher); // true
console.log(chai instanceof User);    // true
console.log(chai instanceof Object);  // true
```

🧠 Memory Hook:

🧬 **Inheritance is like a layered cake** 🎂:

- The top layer (`Teacher`) can eat from the lower layers (`User` and `Object`)
- But the bottom (`Object`) knows nothing about who's on top

🧩 Topic: `static` Methods in JavaScript Classes

☑ Definition:

`static` methods belong to the **class itself**, not to the instances created from that class.

🧠 Analogy: Coffee Machine ☕ vs. Coffee Cup 🍵

- `class` → Coffee Machine
- `instance` → Coffee Cup
- `static` methods → Only work on the **Machine**, not on the individual **Cup**.

```
class CoffeeMachine {
    static powerOn() {
```

```
        console.log("🔌 Machine is ON");
    }

    makeCoffee() {
        console.log("☕ Making coffee...");
    }
}

const cup = new CoffeeMachine();
cup.makeCoffee();       // ☑️ Works
cup.powerOn();          // ❌ Error: Not a function
CoffeeMachine.powerOn() // ☑️ Works
```

👩‍🏭 Your Code:

```
class User {
    constructor(username){
        this.username = username
    }

    logMe(){
        console.log(`Username: ${this.username}`);
    }

    static createId(){
        return `123`;
    }
}

const hitesh = new User("hitesh")
// console.log(hitesh.createId()) ❌

class Teacher extends User {
    constructor(username, email){
        super(username)
        this.email = email
    }
}

const iphone = new Teacher("iphone", "i@phone.com")
console.log(iphone.createId()); // ❌ Error
```

🔍 Why the Error?

```
iphone.createId(); // ❌ TypeError: iphone.createId is not a function
```

- `createId()` is a **static** method on the `User` class.
- That means it can only be called **directly on the class**, not on an instance.

☑ Correct way:

```
console.log(User.createId());     // ☑ works
console.log(Teacher.createId()); // ☑ also works (inherited by subclass)
```

> ❗ You can't do `hitesh.createId()` or `iphone.createId()` because `createId` is not part of the object instance.

🎯 Behind the Scenes

```
function User(username){
    this.username = username;
}

User.prototype.logMe = function(){
    console.log(`Username: ${this.username}`);
}

User.createId = function(){
    return '123';
}

const hitesh = new User("hitesh");
console.log(hitesh.createId()); // ❌ TypeError
console.log(User.createId());   // ☑
```

☑ Use Case for `static`

When should you use `static`?

- When a method **doesn't depend on instance properties**
- When it's a **utility/helper function**
- When it's **shared across all instances**

Example:

```
class MathUtils {
    static add(x, y) {
        return x + y;
    }
}
```

```
console.log(MathUtils.add(5, 3)); // ☑ 8
```

🩹 Inheritance + `static`

Static methods are **inherited**:

```
class Parent {
    static greet() {
        return "Hello from Parent";
    }
}

class Child extends Parent {}

console.log(Child.greet()); // ☑ "Hello from Parent"
```

So this works:

```
console.log(Teacher.createId()); // ☑ 123
```

☑ Fixed Version of Your Code:

```
class User {
    constructor(username){
        this.username = username
    }

    logMe(){
        console.log(`Username: ${this.username}`);
    }

    static createId(){
        return `123`;
    }
}

const hitesh = new User("hitesh");
// ❌ console.log(hitesh.createId()); -- wrong
console.log(User.createId());      // ☑

class Teacher extends User {
    constructor(username, email){
        super(username);
        this.email = email;
```

```
    }
}

const iphone = new Teacher("iphone", "i@phone.com");
// ❌ console.log(iphone.createId()); -- wrong
console.log(Teacher.createId());   // ☑ 123
```

🧠 Memory Hook:

> **Static = Belongs to the Class, not the instance.**

🧬 Just like a **DNA generator** belongs to the **species**, not each **individual**.

🧩 Summary Table

Feature	Static Method	Instance Method
Called on	Class	Object (instance)
Inherited	☑ Yes (via extends)	☑ Yes
Access to `this`	Refers to class	Refers to object
Example	`User.createId()`	`user.logMe()`
Use Case	Helpers / Utilities	Object-specific behavior

📌 Question:

> Are `static` methods available in **inherited classes and their methods**?

☑ Answer:

Yes! `static` methods **are inherited** by subclasses (i.e., child classes). However, they are **not available to instances** of the class.

🧠 Analogy:

Think of `static` methods like a **toolbox in a teacher's lounge**.

- 👩‍🏫 All teachers (child classes) have access to it.
- 👨‍🎓 But students (instances) **do not**.

☑ Let's See it in Code:

```
class User {
    static greet() {
```

```
        return "👋 Hello from User!";
    }
}

class Admin extends User {}

console.log(User.greet());     // ☑ "Hello from User!"
console.log(Admin.greet());    // ☑ "Hello from User!" - inherited

const u1 = new User();
const a1 = new Admin();

console.log(u1.greet());       // ❌ Error: u1.greet is not a function
console.log(a1.greet());       // ❌ Error: a1.greet is not a function
```

📌 Important Clarification:

🔍 What gets inherited?

Property/Method	Available on Subclass	Available on Instance
Static method	☑ Yes	❌ No
Instance method	☑ Yes	☑ Yes

🔄 Behind the Scenes:

```
function User() {}
User.greet = function () {
    return "Hello from User!";
}

function Admin() {}
Admin.__proto__ = User;

console.log(User.greet());    // ☑
console.log(Admin.greet());   // ☑ inherited
```

So `Admin` inherits the static method through the `__proto__` link to `User`.

🧹 Real Use Case:

```
class Utility {
    static generateId() {
        return Math.floor(Math.random() * 10000);
    }
```

```
}

class Logger extends Utility {}

console.log(Utility.generateId());  // ☑
console.log(Logger.generateId());   // ☑ inherited
```

Even though `Logger` doesn't define `generateId()`, it **inherits** it from `Utility`.

🚫 Static Method on Instances?

Nope ❌ — the instance **doesn't** get the static method.

```
const log = new Logger();
console.log(log.generateId()); // ❌ TypeError
```

To fix this, you call it via the class:

```
console.log(Logger.generateId()); // ☑
```

🧠 Final Memory Tip:

> Static = Stays with the **class**, not the **instance** And yes, it **goes down the inheritance chain** like family wealth 💎 (but only to heirs, not everyone).

You're exploring the use of `.bind()` in JavaScript, especially within **class-based DOM event handling**, as seen in frameworks like **React**.

📌 What's Happening?

You're creating a class `React`, attaching an event listener to a button, and using `.bind(this)` to preserve the context of `this`.

```
this.handleClick.bind(this)
```

🧠 Analogy: The Lost Identity Card

Imagine `this` is your **ID card** at work.

- When you **go outside the building** (pass a function somewhere like an event handler), your identity (`this`) gets **lost** unless you bring your ID.
- `.bind(this)` is like **attaching your ID** permanently to your shirt 🎽 so that people know you're still "you" even outside the original place.

🦁 Behind the Scenes (Why `.bind(this)` is Needed?)

When we write:

```
document.querySelector('button').addEventListener('click', this.handleClick);
```

The `handleClick` function loses its **original context**, because it's **called by the button**, not by your class.

So `this.server` becomes `undefined` or throws an error ❌.

☑️ `.bind(this)` Fixes That!

```
.addEventListener('click', this.handleClick.bind(this))
```

This **permanently binds** the method `handleClick` to the class instance (`this`), no matter who calls it.

💡 Here's Your Example, Enhanced with Comments

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
    <title>React Example</title>
</head>
<body>
    <button>Click Me!</button>

    <script>
        class React {
            constructor(){
                this.library = "React"
                this.server = "https://localhost:300"

                // 🧠 Binding 'this' so handleClick knows 'this' refers to the
class instance
                document.querySelector('button')
                        .addEventListener('click', this.handleClick.bind(this))
            }
```

```
            handleClick(){
                console.log("🖱️ Button Clicked!");
                console.log("📡 Server:", this.server);
            }
        }

        const app = new React()
    </script>
</body>
</html>
```

🔍 What If You Don't Use `.bind(this)`?

```
document.querySelector('button')
        .addEventListener('click', this.handleClick) // ❌ `this` is undefined in handleClick
```

Output on click:

```
button clicked
Uncaught TypeError: Cannot read properties of undefined (reading 'server')
```

🔁 Alternative: Use Arrow Function (Lexical `this`)

```
document.querySelector('button')
        .addEventListener('click', () => this.handleClick())
```

- Arrow functions **capture `this` from the surrounding context** (lexical scope), so it's already bound.

🔬 What's `.bind()` Exactly?

➕ Syntax:

```
const newFunc = originalFunc.bind(context)
```

- `context` becomes the `this` inside `originalFunc`
- Returns a new copy of the function

```
const boundFunc = this.handleClick.bind(this)
button.addEventListener('click', boundFunc)
```

🧪 Use Case Summary

Situation	Should you use `.bind()`?
Passing class methods to event listeners	☑ Yes
Inline arrow function usage in short	☑ Can use arrow
Performance critical or reused method	☑ Use `.bind()` once
React class components (not hooks)	☑ Required to bind or use arrow

---

💥 Final Tip:

Always Remember:

> 🔁 **this is dynamic** 📌 Use `.bind(this)` or arrow functions to **lock it** where needed!

🔍 What are Property Descriptors?

Every property in JavaScript has **meta-properties** behind it like:

Property	What It Controls
`value`	The actual value stored 🤔
`writable`	Can the value be changed? ✍️
`enumerable`	Will it show up in loops like `for...in`, `Object.keys`? 🔍
`configurable`	Can it be deleted or redefined? 🧨

You can **view and modify** these using:

- `Object.getOwnPropertyDescriptor(obj, prop)`
- `Object.defineProperty(obj, prop, descriptor)`

🧪 Example Breakdown

◇ Part 1: Getting Descriptors for `Math.PI`

```
const descriptor = Object.getOwnPropertyDescriptor(Math, "PI")
console.log(descriptor)
```

Output:

```
{
  value: 3.141592653589793,
  writable: false,
  enumerable: false,
  configurable: false
}
```

Analogy:

Imagine `Math.PI` is a **sealed book** :

- ❌ You can **read** it (), but can't **edit** it (`writable: false`)
- ❌ It doesn't show up in loops (`enumerable: false`)
- ❌ You can't replace or delete it (`configurable: false`)

Try this:

```
Math.PI = 5
console.log(Math.PI) // still 3.1415...
```

Part 2: Custom Object Example

```
const chai = {
    name: 'ginger chai',
    price: 250,
    isAvailable: true,
    orderChai: function(){
        console.log("chai nhi bni");
    }
}
```

Analogy:

Think of `chai` as a **menu item** . You're checking the **"name" label's properties**.

```
console.log(Object.getOwnPropertyDescriptor(chai, "name"));
```

Default descriptor:

```
{
  value: 'ginger chai',
  writable: true,
  enumerable: true,
  configurable: true
}
```

Now Changing Descriptor

```
Object.defineProperty(chai, 'name', {
    enumerable: true
})
```

This doesn't remove existing properties (`writable` or `configurable`) unless explicitly changed. It just **overrides what's specified**.

Now again:

```
console.log(Object.getOwnPropertyDescriptor(chai, "name"));
```

Output remains mostly unchanged:

```
{
  value: 'ginger chai',
  writable: true,
  enumerable: true,
  configurable: true
}
```

✔ Why? Because you only set `enumerable`, others stay intact.

Looping Through Object

```
for (let [key, value] of Object.entries(chai)) {
    if (typeof value !== 'function') {
        console.log(`${key} : ${value}`);
    }
}
```

This skips methods like `orderChai` (smart filtering!) and prints:

```
name : ginger chai
price : 250
isAvailable : true
```

🧠 Analogy Time: 🛠️ Object.defineProperty

Think of `Object.defineProperty` like a **property customizer tool** at a factory 🔧:

- You can **lock** a property (`writable: false`)
- Hide it from audits (`enumerable: false`)
- Or make it permanent (`configurable: false`)

💡 Use Cases

Use Case	Descriptor to Use
Read-only constants (e.g., `PI`)	`writable: false`
Hidden system/internal properties	`enumerable: false`
Prevent accidental deletion	`configurable: false`

🧪 Bonus: Freeze a Property Completely

```
Object.defineProperty(obj, 'key', {
  writable: false,
  enumerable: false,
  configurable: false
})
```

This makes the property:

- 🧱 **Immutable**
- 🙈 **Invisible**
- 🔒 **Permanent**

🧠 Summary (Emoji Style)

- 📖 `value`: Actual stored value
- ✍️ `writable`: Can it be changed?
- 🔍 `enumerable`: Will it show in loops?
- 🔒 `configurable`: Can it be deleted/modified?

🔚 Final Words

Understanding property descriptors helps you become a **JavaScript internals wizard** 🧙. They're rarely used day-to-day, but absolutely crucial when:

- Writing secure code
- Building frameworks
- Creating libraries with immutability

🚧 Understanding Object Descriptors

You already know:

```
const chai = {
  name: 'ginger chai',
  price: 250,
  isAvailable: true,
  orderChai: function() {
    console.log("chai nhi bni");
  }
}
```

But **behind the scenes**, when you define this object, JavaScript **automatically attaches some metadata** (called **property descriptors**) to each property like `name`, `price`, etc.

🧠 Analogy: Real-World Product Label 🏷️

Imagine each property is a **product on a shelf**. JavaScript doesn't just store the value; it also stores **labels** saying:

- Can you change it? ✍️
- Can you delete it? 🗑️
- Should it appear in an audit? 🔍

These labels are stored **internally in hidden slots** of the object.

🔍 Behind-the-Scenes Example

When you write:

```
const chai = {
  name: "ginger chai"
}
```

The JS engine (like V8) does **something like this internally**:

```
let chai = {}
Object.defineProperty(chai, 'name', {
  value: 'ginger chai',
  writable: true,
  enumerable: true,
  configurable: true
})
```

Yes, **literally like that!** 🎯

📌 What is `Object.defineProperty`?

It's your way of **manually specifying the behavior of a property**. Example:

```
Object.defineProperty(chai, 'name', {
  writable: false,  // cannot change
  enumerable: false, // hidden in loops
  configurable: false // cannot delete or redefine
})
```

🧠 Think of this as **editing the label** on the product!

🔄 So What Happens Internally?

☑️ Step-by-step behind the scenes:

1. The JS engine checks if the object exists (`chai` ☑️)
2. Then checks if the property (`name`) already exists
3. If not, it defines it from scratch using the given descriptors
4. If yes:
    - It checks if `configurable: true`
    - If it's not configurable and you're trying to change descriptors: ❌ it throws an error

---

🧹 `Object.getOwnPropertyDescriptor`

This just **reads the hidden label** of the property:

```
console.log(Object.getOwnPropertyDescriptor(chai, "name"))
```

Returns:

```
{
  value: "ginger chai",
  writable: true,
  enumerable: true,
  configurable: true
}
```

🔧 Relation to `__proto__` and Inheritance

Now let's tie it with inheritance!

Every object has:

- Its **own properties** (like `chai.name`)
- A **prototype link** via `__proto__`

```
const tea = {}
console.log(tea.__proto__ === Object.prototype); // true
```

What is `__proto__`?

🫣 It's a **pointer** (aka internal slot `[[Prototype]]`) that links one object to another to enable **inheritance**.

So if a property isn't found in `chai`, JS says:

> "Hmm, let me check its `__proto__` object (prototype)"

🔁 How JavaScript Resolves Properties

Let's say:

```
const chai = {
  name: "ginger chai"
}

chai.__proto__ = {
  color: "brown"
}

console.log(chai.color); // "brown"
```

➡️ JavaScript does:

1. 🔍 Check if `chai` has property `color`

2. ❌ Not found
3. Go to `chai.__proto__` and look there ☑

🔄 Full Analogy Recap

Concept	Analogy
`value`	📦 Actual content inside the box
`writable`	✍️ Is the label editable?
`enumerable`	🔍 Will it show up in inventory listing?
`configurable`	🔒 Can you remove or re-label the item?
`__proto__`	🧬 The genetic parent or blueprint of an object

🧪 Advanced Hidden Behavior (for Nerds 🧠)

Hidden Internal Property Table

Internally, every object looks like this to the engine:

```
chai = {
  [[Prototype]]: Object.prototype,
  [[Properties]]: {
    name: {
      [[Value]]: "ginger chai",
      [[Writable]]: true,
      [[Enumerable]]: true,
      [[Configurable]]: true
    }
  }
}
```

☑ These `[[...]]` are called **"internal slots"** — not accessible directly, but used by the engine.

🧠 Pro Developer Tip

Want to make a property:

- 🔒 Immutable
- 🙈 Invisible
- ⛔ Non-deletable

Use:

```
Object.defineProperty(obj, "secret", {
  value: "💣",
  writable: false,
  enumerable: false,
  configurable: false
})
```

This makes it **bulletproof** 🔐.

💬 Final Example with All Concepts

```
const user = {
  name: "Darshan",
  email: "darshan@chai.com"
}

// Freeze the name:
Object.defineProperty(user, "name", {
  writable: false,
  enumerable: false,
  configurable: false
})

console.log(Object.getOwnPropertyDescriptor(user, "name"));

// Try to change name
user.name = "Someone"
console.log(user.name); // Still Darshan

// Try to delete
delete user.name
console.log(user.name); // Still Darshan
```

☑️ Summary:

- 🛠️ `Object.defineProperty()` customizes property behavior.
- 🕵️ `Object.getOwnPropertyDescriptor()` lets you inspect hidden property flags.
- 🧬 `__proto__` is the internal inheritance chain link.
- 💧 Use descriptors to make properties read-only, hidden, or permanent.

❓ Can You Change a `const` Object's Property?

☑️ **Yes, you can change the properties of a `const` object.** ❌ But you cannot reassign the object reference itself.

🔍 What does `const` actually do?

When you declare an object with `const`, you're saying:

> "This **binding** (reference) to the object cannot be changed."

You're **not** saying the object itself is frozen or immutable.

🧪 Example: Mutating Properties of a `const` Object

```
const car = {
  brand: "Tesla",
  model: "Model 3"
}

// ☑️ You can change a property
car.model = "Model S"
console.log(car.model) // Output: Model S

// ☑️ You can add a new property
car.year = 2024
console.log(car.year) // Output: 2024

// ❌ You cannot reassign the whole object
car = { brand: "BMW" } // ❌ TypeError: Assignment to constant variable.
```

🧠 Analogy Time

Imagine `const` is like assigning a **locker key** 🔑:

- You're given a locker (`const car = {...}`)
- You **can put things in it, rearrange them**, or even remove things.
- But you **can't throw away the locker and replace it with a new one**.

💡 Want to Truly Prevent Changes?

Use:

```
Object.freeze(car) // Makes object fully immutable (shallow freeze)
```

Or to freeze deeply:

```
function deepFreeze(obj) {
  Object.freeze(obj)
  for (let key in obj) {
```

```
      if (
        typeof obj[key] === "object" &&
        obj[key] !== null &&
        !Object.isFrozen(obj[key])
      ) {
        deepFreeze(obj[key])
      }
    }
  }

  deepFreeze(car)
```

Now any property update or addition will fail silently in non-strict mode or throw in strict mode.

☑ Summary

Action	**Possible with `const` object?**
Change property value	☑ Yes
Add new property	☑ Yes
Delete a property	☑ Yes (unless frozen)
Reassign object (e.g., `obj = {}`)	❌ No
Freeze object to prevent changes	🔒 Use `Object.freeze()`

✨ What are Getters and Setters in JavaScript?

> Getters and setters are *special methods* that help you **control access to object properties**. They are used to get (retrieve) or set (update) the values of properties with extra logic, while appearing like simple property access.

📦 Default vs User-defined Properties

Aspect	**Default Property**	**User-defined Getter/Setter**
Access	Directly stores and retrieves value	Runs custom logic on get/set
Syntax	`this.prop = value`	`get prop() {}` / `set prop(val) {}`
Control	❌ No control on access	☑ Can validate, transform, or hide data
Privacy	❌ Exposed	☑ Controlled with conventions like `_prop`
Return Value	Just value	Getter must return, Setter can't return

---

🔍 Let's Break it Down With Examples

🧾 Example 1: Using ES6 Class Syntax

```
class User {
    constructor(email, password){
        this.email = email;      // Setter is triggered
        this.password = password // Setter is triggered
    }

    get email(){
        return this._email.toUpperCase(); // Get transformed value
    }

    set email(value){
        this._email = value; // Save raw value internally
    }

    get password(){
        return `${this._password}hitesh`; // Add suffix on access
    }

    set password(value){
        this._password = value;
    }
}

const hitesh = new User("h@hitesh.ai", "abc")
console.log(hitesh.email); // H@HITESH.AI
console.log(hitesh.password); // abchitesh
```

⚠️ What if you used `this.email = email` inside setter?

It causes a **Maximum Call Stack Exceeded** error! 💀 That's because:

- `this.email = value` calls the setter again (since setter is named `email`)
- Infinite loop 🔁

☑️ **Solution:** Use a backing property like `_email`.

```
set email(value) {
  this._email = value; // avoids recursion
}
```

🧠 Behind the Scenes Analogy

> Think of a **Getter as a receptionist** 🤵 who fetches your request. Think of a **Setter as a gatekeeper** 🛡️ who verifies and stores the data.

🧪 Example:

- You ask: "Give me the email."
- The **getter** transforms it to uppercase before showing you.
- When you say: "Set email to xyz", the **setter** stores the original as `_email`.

Version 2: Using `Object.defineProperty()`

```
function User(email, password){
    this._email = email;
    this._password = password;

    Object.defineProperty(this, 'email', {
        get: function(){
            return this._email.toUpperCase(); // logic on get
        },
        set: function(value){
            this._email = value;
        }
    });

    Object.defineProperty(this, 'password', {
        get: function(){
            return this._password.toUpperCase(); // can encrypt, mask, etc.
        },
        set: function(value){
            this._password = value;
        }
    });
}

const chai = new User("chai@chai.com", "chai")
console.log(chai.email); // CHAI@CHAI.COM
```

Behind the Scene

- `Object.defineProperty()` creates a **hidden layer** of getter/setter.
- Allows you to define behavior without rewriting the object/class structure.

Version 3: Object Literal Pattern

```
const User = {
    _email: 'h@hc.com',
    _password: "abc",

    get email(){
        return this._email.toUpperCase();
    },
```

```
    set email(value){
        this._email = value;
    }
}

const tea = Object.create(User);
console.log(tea.email); // H@HC.COM
```

⚖️ With Getter/Setter vs Without

🔧 Without:

```
const user = {
  email: 'abc@xyz.com'
};

console.log(user.email); // raw value
```

🚀 With:

```
const user = {
  _email: 'abc@xyz.com',

  get email() {
    return this._email.toUpperCase();
  },

  set email(value) {
    this._email = value.trim(); // validation logic
  }
};
```

🔐 ES2022 Private Fields

```
class User {
    #email;
    constructor(email){
        this.#email = email;
    }

    get email(){
        return this.#email.toUpperCase();
    }
```

```
    set email(value){
        this.#email = value;
    }
}

const user = new User("private@email.com");
console.log(user.email); // PRIVATE@EMAIL.COM
```

☑ Benefits:

- True encapsulation 🔒
- Only accessible within the class
- Can't be accessed or modified outside

🧠 Getter/Setter vs Memory 📦

> Think of memory as **storage units**.

- **Setters** move values **into** memory (like storing into a locker)
- **Getters** fetch the value **from** memory (like retrieving from the locker)

Example:

```
this._name = "darshan"; // goes to memory
get name() { return this._name.toUpperCase(); } // comes from memory with change
```

📐 Real World Use Cases

- ☑ Formatting data before displaying (like `.toUpperCase()`)
- ☑ Input validation (e.g., trimming, checking length)
- ☑ Creating computed properties (`fullName`, etc.)
- ☑ Logging changes (e.g., audit logs)
- ☑ Masking sensitive data (e.g., password masking)
- ☑ Working with virtual fields in MongoDB schemas

📏 Array `.length` – a Getter/Setter?

Yes! ☑ When you do:

```
let arr = [1, 2, 3, 4];
console.log(arr.length); // Getter

arr.length = 2; // Setter
console.log(arr); // [1, 2]
```

🧠 Internally:

- `.length` is a **property with special getter and setter**
- Setting `length` auto-truncates or expands the array

📌 Summary Chart

Concept	Explanation
`get`	Used to *return* custom logic on property access
`set`	Used to *set* or validate incoming data
Naming	Should avoid using same name inside setter — use `_` prefix
Must Return	Getter must return something
Must Not Return	Setter should not return anything
ES6	Supports in classes using `get/set` keywords
Old-style	Use `Object.defineProperty()`
Objects	Literal syntax also supports getter/setter
ES2022	True private fields using `#property`
Memory	Set = save to memory, Get = read from memory

👩‍🏫 Conclusion

> **Getter and Setter** are *power tools* in JavaScript that help you design better APIs, ensure data safety, and control object behavior with elegance.

🔑 They allow:

- Data encapsulation
- Clean syntax
- Flexible logic injection
- Enhanced maintainability

🏁 What is a Race Condition?

🏃 **Race Condition** occurs when multiple operations are executed in **unpredictable order**, and the final outcome depends on the timing or sequence of these operations. This can lead to bugs or unexpected behavior, especially in asynchronous environments.

Example scenario:

- Two or more asynchronous operations (like requests or data manipulations) might **interfere with each other**, causing an inconsistent state in the system.

- In a **synchronous** example, if one operation depends on the results of another, and the order is not guaranteed, we can encounter unexpected issues.

Example of Race Condition:

```
class User {
    constructor(email, password){
        this.email = email;  // This line triggers the setter (email)
        this.password = password;  // This line triggers the setter (password)
    }

    get email() {
        console.log('Getting Email');
        return this._email.toUpperCase();
    }

    set email(value) {
        console.log('Setting Email');
        this._email = value;
    }

    get password() {
        console.log('Getting Password');
        return `${this._password}hitesh`;
    }

    set password(value) {
        console.log('Setting Password');
        this._password = value;
    }
}

const user = new User('test@example.com', 'myPassword');
```

In this code:

- When we create a new instance of `User`, both the **setter methods** for `email` and `password` are called.
- These **setters** (because they are designed to trigger the **getter**) would call each other recursively if not designed properly, leading to a **Maximum Call Stack Size Exceeded** error.

⚠ Maximum Call Stack Size Exceeded (Stack Overflow Error)

The **stack overflow error** occurs due to **recursive calls** in setter or getter functions. When the setter or getter tries to call itself (or another function that eventually calls itself), it leads to **infinite recursion**.

In Our Case (Stack Overflow)

Here's how the **getter and setter** design leads to this issue:

```
class User {
    constructor(email, password){
        this.email = email;  // Setter called here
        this.password = password; // Setter called here
    }

    get email(){
        return this._email.toUpperCase(); // Getter accesses '_email' but triggers
setter
    }

    set email(value){
        this._email = value;  // This setter sets '_email'
    }
}
```

The problem:

- The **setter** for `email` assigns the value to `this._email`. But **inside the constructor**, when we call `this.email = email`, we actually invoke the **setter**.
- The **getter** for `email` is triggered when `this._email.toUpperCase()` is accessed, but it again calls the setter because `this.email` is referenced inside the getter.

This causes **infinite recursion**:

1. `this.email = email` triggers the **setter**.
2. The **setter** tries to assign to `this._email`, and on accessing `this.email` (via the **getter**), the **setter** is triggered again, and the cycle repeats endlessly.

🔧 How to Solve This (Fixing Stack Overflow)

☑ Solution: Use a Backing Property with an Underscore `_email`

- **Back to the Basics**: To **break the recursion**, use a backing property to store the value.
- Instead of setting `this.email` directly in the setter, store the value in a hidden property (e.g., `_email`), which **prevents the getter from calling the setter** again.

🚀 Updated Solution Example

```
class User {
    constructor(email, password){
        this.email = email;  // Trigger the setter correctly
        this.password = password;  // Trigger the setter correctly
    }

    get email(){
        return this._email.toUpperCase();  // Return transformed value, not
triggering setter
```

```
    }

    set email(value){
        this._email = value;  // Store the raw value in a hidden property
    }

    get password(){
        return `${this._password}hitesh`;  // Return password with a suffix
    }

    set password(value){
        this._password = value;  // Store raw password value
    }
}

const hitesh = new User("h@hitesh.ai", "abc");
console.log(hitesh.email); // H@HITESH.AI
```

How This Solves the Problem:

- The **setter** now stores the raw email in a private backing variable (`_email`), so the **getter** doesn't trigger the setter again.
- The **getter** for `email` only transforms the value for display but doesn't access the setter again, thus **breaking the recursion** and **avoiding the stack overflow**.

🎯 How the Solution Works (Behind the Scenes)

- **Constructor**:
    - When you create a new `User` instance with `new User(email, password)`, the constructor initializes the values and uses the **setter** to store values in the private variables `_email` and `_password`.
- **Setter**:
    - The setter **does not call the getter** anymore. Instead, it stores the value directly in `_email` and `_password`, ensuring that no infinite recursion occurs.
- **Getter**:
    - When you access `user.email`, the getter method is invoked. It only reads the value of `_email` and applies the transformation (`toUpperCase()`), but does not call the setter for `email`. This way, the setter is **only invoked once**, and the getter doesn't trigger unnecessary recursion.

🧩 Race Condition Solution:

In the **getter/setter context**, race conditions could occur when multiple asynchronous or synchronous operations are trying to update the same properties at the same time.

Potential problem:

- Suppose multiple processes or threads (in case of multi-threaded environments) are trying to update `email` or `password`. If the setter for `email` or `password` is not handled properly, we may encounter inconsistent values.

Solution:

- Using **private backing properties** (`_email`, `_password`) to avoid direct modification of `email` and `password` ensures that only the controlled methods (getter/setter) can update and fetch the values.
- This approach can help **reduce the risk of race conditions**, as it ensures **single-threaded access** to properties.

 Conclusion

Race Condition and Stack Overflow in Getters/Setters:

- A **stack overflow** happens when getters/setters call each other recursively without proper handling, causing infinite recursion.
- **Race conditions** can occur when multiple operations try to access/update shared properties simultaneously. This is controlled by using private properties, ensuring proper synchronization and data encapsulation.

Key Points to Remember:

- **Back-end Encapsulation**: Use backing variables (e.g., `_email`) to store values internally.
- **Getter/Setter Logic**: **Return values** in getters; **set** values in setters without returning anything.
- **Avoid Infinite Recursion**: Never reference the same property inside its getter or setter.
- **Race Condition Prevention**: Use private variables and controlled access through getter/setter methods to prevent external modifications or interference.
