**Prototype-**

In JavaScript, every object has a hidden internal property called [[Prototype]] (or \_\_proto\_\_ in some environments). This prototype is another object from which the original object inherits methods and properties.

This forms the basis of prototypal inheritance.

🔹 Why is Prototype Used?

To allow objects to share behavior (methods) without copying them into every instance.

🔹 Understanding with Example

function Person(name, age) {

this.name = name;

this.age = age;

}

// Adding a method to Person's prototype

Person.prototype.sayHello = function() {

console.log(`Hello, my name is ${this.name}`);

};

// Creating a new object

const person1 = new Person("Alice", 30);

// Accessing sayHello method via prototype

person1.sayHello(); // Output: Hello, my name is Alice

⚙ How it works:

person1 does not have sayHello method directly on it.

JavaScript looks up the prototype chain and finds sayHello in Person.prototype.

1.All JavaScript objects inherit properties and methods from a prototype.

**Prototype Inheritance**

All JavaScript objects inherit properties and methods from a prototype:

Date objects inherit from Date.prototype

Array objects inherit from Array.prototype

Person objects inherit from Person.prototype

The Object.prototype is on the top of the prototype inheritance chain:

Date objects, Array objects, and Person objects inherit from Object.prototype.

Adding Properties and Methods to Objects

Sometimes you want to add new properties (or methods) to all existing objects of a given type.

Sometimes you want to add new properties (or methods) to an object constructor.

Using the prototype Property

The JavaScript prototype property allows you to add new properties to object constructors:

Example

function Person(first, last, age, eyecolor) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eyecolor;

}

Person.prototype.nationality = "English";

The JavaScript prototype property also allows you to add new methods to objects constructors:

Example

function Person(first, last, age, eyecolor) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eyecolor;

}

Person.prototype.name = function() {

return this.firstName + " " + this.lastName;

};

**What is Prototype Inheritance?**

When one object **inherits properties or methods** from another object via its prototype.

Example-

function Animal(name) {

this.name = name;

}

Animal.prototype.eat = function() {

console.log(`${this.name} is eating`);

};

function Dog(name, breed) {

Animal.call(this, name); // Inherit properties

this.breed = breed;

}

// Inherit methods using prototype

Dog.prototype = Object.create(Animal.prototype);

Dog.prototype.constructor = Dog; // Reset constructor

Dog.prototype.bark = function() {

console.log(`${this.name} is barking`);

};

const dog1 = new Dog("Rex", "Labrador");

dog1.eat(); // Inherited from Animal

dog1.bark(); // Own method

### What's Happening:

* Dog inherits from Animal.
* dog1 can use both eat() and bark() methods.
* JavaScript follows the prototype chain:
  + dog1 → Dog.prototype → Animal.prototype

## 🔹 \_\_proto\_\_ vs prototype

| **Term** | **Meaning** |
| --- | --- |
| \_\_proto\_\_ | The actual internal prototype reference of an object. |
| prototype | A property of constructor functions that becomes the \_\_proto\_\_ of instances created with new. |

function Car() {}

const car1 = new Car();

console.log(car1.\_\_proto\_\_ === Car.prototype); // true

## Summary

* Every JavaScript object has a prototype.
* Prototypes allow inheritance of methods and properties.
* Prototypes reduce memory usage and enable code reuse.
* Object.create(), new, and class are ways to set up prototype chains.

animal

┌───────────────┐

│ eatsFood: true│

└───────────────┘

▲

│ [[Prototype]]

│

┌───────────────┐

│rabbitJumps: true│

└───────────────┘

Rabbit

 rabbit has its own property: rabbitJumps: true

 rabbit's **prototype** is set to the animal object.

 So, when we access rabbit.eatsFood, JavaScript doesn’t find it in rabbit, so it **goes up the prototype chain** and finds it in animal.

**Approach 1: Using \_\_proto\_\_ (Old & Not Recommended)**

let animal = {

animalEats: true,

};

let rabbit = {

rabbitJumps: true,

};

// Set prototype manually

rabbit.\_\_proto\_\_ = animal;

console.log(rabbit.animalEats); // true ← inherited

console.log(rabbit.rabbitJumps); // true ← own property

🔸 \_\_proto\_\_ directly sets the prototype chain.  
🔸 But it's **deprecated** (shouldn’t be used in production) — only for learning or debugging.

Approach 2: Using Object.setPrototypeOf() (Modern)

let rabbit = {

rabbitJumps: true,

};

let animal = {

animalEats: true,

};

// Set prototype using modern method

Object.setPrototypeOf(rabbit, animal);

console.log(rabbit.animalEats); // true

console.log(rabbit.rabbitJumps); // true

🔹 Same behavior, but cleaner and safer. This is the **modern and recommended way** for dynamically setting prototypes.

## Another Custom Example (Real-World Analogy)

### 👩‍🏫 Let's say:

* Person object has common methods (like greet)
* Student object has its own property (studies), and inherits from Person

let person = {

greet() {

console.log("Hello! I am a person.");

},

};

let student = {

studies: true,

};

// Set person as prototype of student

Object.setPrototypeOf(student, person);

console.log(student.studies); // true (own property)

student.greet();

// Hello! I am a person. (inherited method)

### ✅ What happens here:

* JavaScript looks for greet() on student → not found
* Then it checks student.\_\_proto\_\_ (which is person) → found!
* So greet() is executed from person

## 🚫 Caution

* Avoid deep prototype chains — it makes debugging harder.
* Avoid using \_\_proto\_\_ in real code; use Object.create() or Object.setPrototypeOf().

## Recommended Way to Set Prototype at Creation:

let person = {

greet() {

console.log("Hello!");

},

};

let student = Object.create(person); // Sets prototype at creation

student.studies = true;

student.greet(); // Hello!

Prototype inheritance in JavaScript allows objects to inherit properties and methods from other objects. Each object in JavaScript has an internal link to another object called its prototype. This chain of prototypes forms the prototype chain.

When you access a property or method on an object, JavaScript first checks the object itself. If the property or method isn’t found, it moves up the prototype chain until it finds the property or reaches the end of the chain (null).



const parent = {

greet: function () {

console.log("Hello from the parent object!");

}

};

​

const child = Object.create(parent);

​

child.sayHi = function () {

console.log("Hi from the child object!");

};

​

child.greet();

child.sayHi();

**Output**

Hello from the parent object!

Hi from the child object!

### Prototype Chain

The prototype chain is the mechanism that JavaScript uses to resolve properties and methods. If an object doesn’t have a requested property, the JavaScript engine searches up the prototype chain.



function Animal(name) {

this.name = name;

}

​

Animal.prototype.speak = function () {

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

};

​

const dog = new Animal("Buddy");

​

console.log(dog.name);

dog.speak();

**Output**

Buddy

Buddy makes a sound.