14 – Prototypes

JavaScript is an object-oriented language. This is because all data types in JavaScript are either objects or behave like one. Prototypes are the mechanism by which JavaScript objects inherit features from one another.

**Prototypal Inheritance/Chaining**

Every data type has a base class from which instances of that class are created. For example, every array is an instance (object) of the base class Array. The .prototype is a property of all constructor functions (e.g., Array.prototype), which consists of all its properties and methods. The \_\_proto\_\_ property of the instance points towards the .prototype of its parent class.

Every JavaScript entity, such as Numbers, Strings, Arrays, etc., has an internal hidden property by the name of \_\_proto\_\_ which points to its parent object that it inherits features from. If a property is not found in the object itself, JavaScript looks up the prototype chain, i.e., the object linked via the \_\_proto\_\_ property.

The \_\_proto\_\_ can be accessed by the method Object.getPrototypeOf(objName)

In the case of JavaScript classes, the keyword “extends” is used to signify that a child class inherits the attributes of a parent class. However, this can also be achieved by pointing the \_\_proto\_\_ property of the child class to the parent class.

This concept, wherein JavaScript objects can inherit properties and methods from other via the \_\_proto\_\_ property, is known as Prototypal Inheritance.

For example, let’s say an array by the name of arr is declared.

arr.\_\_proto\_\_ = Array.prototype

arr.\_\_proto\_\_.\_\_proto\_\_ = Object.prototype

arr.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_ = null

Prototype Chain: arr 🡪 Array.prototype 🡪 Object.prototype 🡪 null

const animal = {

eats: true,

walk() {

console.log("Animal walks");

}

};

const dog = {

barks: true

};

dog.\_\_proto\_\_ = animal; // Set animal as the prototype of dog

console.log(dog.barks); // true (own property)

console.log(dog.eats); // true (inherited from animal)

dog.walk(); // "Animal walks"

However, the recommended and cleaner way of creating an object and setting its \_\_proto\_\_ to the object you specify is by using the Object.create() method.

const animal = {

eats: true,

walk() {

console.log("Animal walks");

}

};

const dog = Object.create(animal); // create dog with animal as prototype

dog.barks = true;

console.log(dog.eats); // true (inherited)

dog.walk(); // "Animal walks"

console.log(dog.\_\_proto\_\_ === animal); // true

**Do’s and Don’ts While Modifying Prototypes**

1) Use Prototypes for Shared Methods

Define methods on the prototype instead of the constructor itself to save memory.

function User(name) { // Not defining shared method in the constructor itself

this.name = name;

}

User.prototype.sayHello = function() { // Defining in the .prototype of the constructor

console.log(`Hi, I'm ${this.name}`);

};

In the above, all User instances will inherit the sayHello() method.

2) Use Object.getPrototypeOf() and Object.setPrototypeOf() instead of \_\_proto\_\_

Object.getPrototypeOf(obj);  
Object.setPrototypeOf(obj, newProto);

For safer code that is to be used as intended.

3) Do NOT modify built-in prototypes (Array, String, etc.)

May lead to performance issues.

4) Do NOT use \_\_proto\_\_ directly

Although it is supported, its use has been deprecated as it causes performance issues.