Class 26 - OOPS 3 - Prototype & Prototypal Inheritance

Agenda

Prototype Object

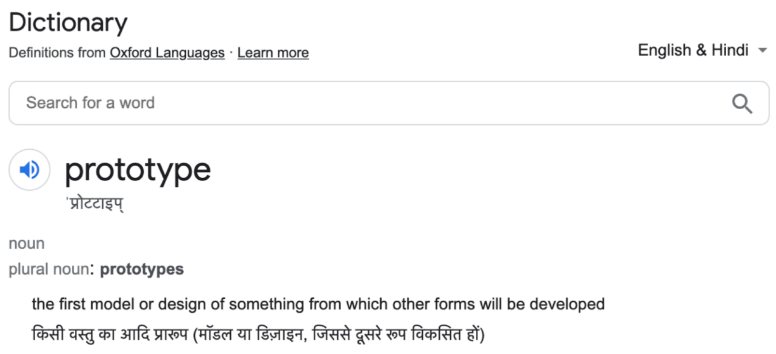
Prototypal Inheritance

proto property

Prototypal Chaining

Object.create()

Prototype Intuition

- 

Dictionary

Definitions from Oxford Languages · Learn more

English & Hindi

Search for a word

prototype

ˈप्रोटटाइप्

noun

plural noun: **prototypes**

the first model or design of something from which other forms will be developed

किसी वस्तु का आदि प्रारूप (मॉडल या डिज़ाइन, जिससे दूसरे रूप विकसित हों)

Pretext

Pretext

In programming we often want to take something and extend it

Like we may have user object and now we may want admin or employees
JavaScript as we know doesn't have classes in true sense

Prototypes in 4 points

1. Complimentary Dish

1. Taj restaurant anecdote where they offered a complimentary dish
2. Similarly there are few methods and properties that Javascript provides to us as a complimentary stuff

```
const obj = {}
console.log(obj.toString())
```

3. The thing to notice is not the output but the fact that we were able to call a method (toString) without even defining it anywhere
4. Just as the restaurant gave me the complimentary dish to enhance my dining experience, similarly JS is giving me few things to enhance my dev experience
5. Lets see one more example

```
const arr = [1,2,3]
console.log(arr.join("->"))
```

6. Again here, we did not define the join method anywhere but it is available and accessible to us for our use

What is Prototype

1. Prototype is basically an object that has methods and properties that gets attached to your object
2. Prototypes are the mechanism by which JavaScript objects inherit features from one another.

2. Mom’s snacks

1. During college days, when the holiday break was over, mom would prepare snacks for hostel based on our liking
2. Similarly JS knows what kind of prototype would work in what scenario

```
const user = {
   name:'Kohli',
   age: 34
}
console.log(user)
```

3. See the console in the browser and check the prototype object

```
const arr1 = [1,2,3]
console.log(arr1);
```

4. Notice the array prototype in the console
5. In JavaScript, objects have a special hidden property [[Prototype]] (as named in the specification), that is either null or references another object. That object is called “a prototype”:
6. The object referenced by [[Prototype]] is called a “prototype”.

3. Family is given but friends are chosen

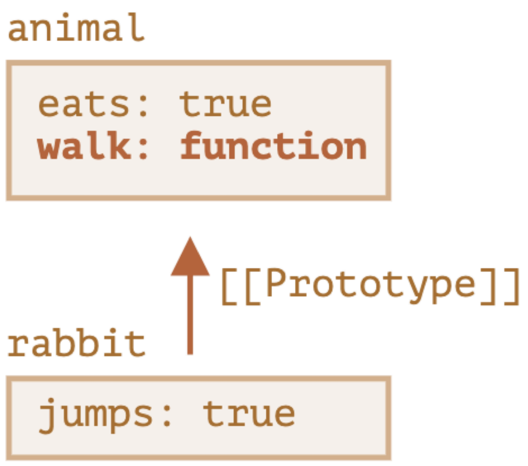
1. We are born to a family and we do not have a say in who our parents and siblings are.
2. But we can choose whom we want to friends with
3. Similarly in JS, we have a prototype that is available by default but at the same time, we also have option to choose what should be our prototype object
4. What is the benefit of having friends
    a. Suppose we want a jacket for a party and we know that our friend has a nice one
    b. Depending on how good a friends you guys are, you can ask for that jacket or simply pick it and use it for your need
    c. Sometimes it is the case that our friends might not help but they might introduce us to their friends who can be of help
5. Similarly in JS, the property [[Prototype]] is internal and hidden but there are ways to set it.
6. One of the way is to use the special name __proto__

```
const animal = {
   eats: true,
   sleep: true,
   walk(){
       console.log('the animal walk')
   }
}
console.log(animal);

const rabbit = {
   areCute: true
}

rabbit.__proto__ = animal
console.log(rabbit);
```

7. Now if we try to access the walk method on rabbit, it will be available

```
console.log(rabbit.walk());
```

8. So if animal has a lot of useful properties and methods, then they become automatically available in rabbit.
9. Such properties are called “inherited”.



10.

PROTOTYPAL CHAIN: LONGER VERSION

```


const animal = {
   eats: true,
   sleep: true,
   walk(){
       console.log('the animal walk')
   }
}
console.log(animal);

const herbivore = {
   eatsMeat: 'naah',
   __proto__: animal
}

const carnivore = {
   eatsMeat: 'yes',
   __proto__: animal

}

const rabbit = {
   canJump: true,
   __proto__: herbivore
}

const tiger = {
   canKill: true,
   __proto__: carnivore
}



console.log(rabbit.eatsMeat);
console.log(tiger.eatsMeat);
```

Let us look for a property that does not exist

```
console.log(rabbit.dance);
```

The idea is that JS first searches for the key in that object , then it goes to its prototype then its prototype all the way till a null object is reached

If it finds a matching key, it does an early return else keeps on searching recursively in prototypes

Prototypal chain

```
console.log(rabbit.__proto__.__proto__.__proto__.__proto__)
```

4. God Element

1. Like there is a common spark in all of us, in JS at the base of everything lies the Object prototype
2. See that object, arrays, string, all of them have Object as their prototype
3. This is the reason why we hear that everything in JS is an object

Traversing through properties

1. Object.keys
    a. `console.log(Object.keys(rabbit))`
    b. Object.keys only return its own keys
2. For .. in loop

```
for(let key in rabbit){
   console.log(key)
}
```

    a. for .. in loop returns both self and inherited properties

3. Using the hasOwnProperty Method
    a. The hasOwnProperty method in JavaScript is used to check whether an object has a particular property as a direct property or not

```
for(let key in rabbit){
   if(rabbit.hasOwnProperty(key)){
       console.log(key)
   }
}
```

Adding methods on prototype

```

function User(name){
   this.name = name
   this.msg = function(){
       console.log(`Hello ${this.name}!`)
   }
}

const user1 = new User('Kohli')
const user2 = new User('Dhoni')

console.log(user1.msg === user2.msg)
```

1. Each object created like this is creating a separate copy fo the msg function
2. When methods are added directly to the constructor, each instance of the object gets its own copy of the method, leading to higher memory usage.

3. Here user1 and user2 each have their own separate msg method in memory, which is inefficient.
4. This is memory wastage and violates DRY principle
5. What is the better way to do this - using prototypes
6. So every time A new Object will be created by the Constructor Function , msg will be available to it inside the prototype only

```javascript
function BetterUser(name){
   this.name = name
}

BetterUser.prototype.msg = function(){
   console.log(`Hello ${this.name}!`)
}

const user3 = new BetterUser('Shubman')
const user4 = new BetterUser('Rohit')

console.log(user3.msg === user4.msg)
```

7. Defining methods on the prototype allows all instances to share a single copy of the method. This not only saves memory but can also lead to better runtime performance because there is less overhead in creating each new object. All objects created from the BetterUser constructor share the same msg function, making it more memory-efficient.
8. Code maintainability improves because all modifications to the prototype affect all instances. This uniform behavior ensures that changes are centralized and consistent across all object

instances, which is crucial in larger codebases or when updates and maintenance are required.

9. Conclusion - Defining methods on the prototype is generally a better practice in JavaScript when creating constructor functions that produce multiple instances. T

Other built in prototypes

1. Other built-in objects such as Array, Date, Function and others also keep methods in prototypes.
2. For instance, when we create an array [1, 2, 3], the default new Array() constructor is used internally. So Array.prototype becomes its prototype and provides methods.

Primitives

1. Primitives are not objects
2. How do we get methods for them
    a. toLowerCase, substring for String
    b. toFixed etc for Number
    c. toString, valueOf for Boolean
3. These primitives also have their prototype defined
    a. String.prototype
    b. Number.prototype
    c. Boolean.prototype
4. Let us try to add a custom method on string so that every new created string can have this custom method

5. Requirement
    a. Create a custom method on string
    b. It accepts any pattern as input and modified the given string
    c. Eg for i/p string “scaler” with pattern as “->”, the output is “s->c->a->l->e->r”
6. Solution
    a. Since we want a method on all the strings, we will add the method on the prototype

```
String.prototype.crazyMethod = function(pattern){
   return this.split("").join(pattern)
}
console.log("scaler".crazyMethod("->"))
console.log("scaler".crazyMethod("*"))
console.log("scaler".crazyMethod("<3"))
```

title: Object.create

1. There is one more way in which we can achieve Prototypal Inheritance with a more direct approach that is by using Object.Create() method
2. Object.create() is a powerful method in JavaScript that allows you to create new objects with a specified prototype object and properties.
3. This method provides a direct form of prototypal inheritance. Here's a step-by-step breakdown to understand how Object.create() works.

4. **The Object.create() method creates a new object and directly sets the new object's prototype to the object that is passed as its first argument.**
5. Let's apply Object.create() to create instances of cars, where each car can share a common set of behaviors (methods) defined in a prototype.
   a. Step 1: Define the Prototype Object
      i. First, we define an object that contains the methods that our car instances will inherit. This object will serve as the prototype for every car instance we create.

```
const carPrototype = {
   displayInfo: function() {
     return `This is a ${this.year} ${this.model}.`;
   }
 };
```

      ii. This carPrototype object has one method, displayInfo, which returns a string containing the car's year and model. The properties year and model are not defined in carPrototype because these will vary from instance to instance.
   b. Step 2: Create Instances Using Object.create()
      i. Now, we use Object.create() to create new car objects that inherit from carPrototype.

```
const car1 = Object.create(carPrototype);
 car1.model = "Toyota Camry";
 car1.year = 2021;
  const car2 = Object.create(carPrototype);
 car2.model = "Ford Mustang";
```

```
car2.year = 2023;
```

ii. Here, car1 and car2 are created with carPrototype as their prototype. After creating each car object, we assign specific values to their model and year properties.

c. Step 3: Use the Inherited Method

i. Now that car1 and car2 are set up with their properties, we can call the inherited displayInfo method:

```
console.log(car1.displayInfo()); // "This is a 2021 Toyota Camry."
console.log(car2.displayInfo()); // "This is a 2023 Ford Mustang."
```

5. Conclusion - Object.create() provides a clean and efficient way to set up inheritance in JavaScript. It allows for the creation of objects with a specific prototype, enabling shared behavior across objects while allowing each object to have its own individual properties. This method is particularly useful in cases where objects are similar but do not require the complexity of a constructor function.

Tricky questions

1. Question 1: Setting __proto__ to null

```
const obj = { name: 'Sample Object' };
console.log(obj.toString());  // Normally works

// Set prototype to null
```

```
obj.__proto__ = null;
console.log(obj.toString());  // This will throw an error
// try {
//     console.log(obj.toString());  // This will throw an
error
// } catch (e) {
//     console.log("Error:", e.message);
// }

// There is no built-in way to restore the original prototype
once it's set to null.
```

Soln - Setting __proto__ to null detaches the object from its prototype chain. This means it loses access to methods defined in Object.prototype, such as toString() and hasOwnProperty().

2. Question 2: Constructor Property Manipulation
   To understand this question, let us see one quick thing first

```
function Animal(){

}
console.log(Animal.prototype.constructor)
```

The prototype property is a feature found only on functions, particularly relevant to functions designed to act as constructors.
Every function has a property “prototype”, which includes a constructor property. This constructor property points back to the function itself.

The constructor property on a function's prototype is primarily a reference back to the function (or constructor) that created an instance.

It allows instances of objects to know which constructor function was used to create them. This can be useful if you need to check the origin of an object or recreate new instances similar to an existing object.

```
function Animal() {}
Animal.prototype.speak = function() {
 console.log("Sound!");
};

function Dog() {}
Dog.prototype = Object.create(Animal.prototype);

const dog = new Dog();
console.log(dog); // true
console.log(dog.constructor.name); // Animal
```

After setting up inheritance, Dog.prototype.constructor points to Animal because Dog.prototype was explicitly set to an object created from Animal.prototype.

Soln -
The output above is misleading because dog is actually an instance of Dog, not Animal. The constructor reference should correctly point to Dog:

```
const dog = new Dog();
console.log(dog); // true
Dog.prototype.constructor = Dog;
console.log(dog.constructor.name); // Dog
```

Point to remember here - It's important to maintain accurate references in the constructor property when setting up inheritance chains in JavaScript.

This ensures that objects created from these functions correctly identify their constructor, which aids in type checking and creating further instances of the same type.

Always remember to reset the constructor property when you modify the prototype chain to maintain integrity and clarity in your code.

3. What will be the output

```
function Rabbit() {}
Rabbit.prototype = {
 eats: true,
};

let rabbit = new Rabbit();

Rabbit.prototype = {};

console.log(rabbit.eats); // ? -> true
```

Soln - true

When we assign a new object to the prototype (e.g., Rabbit.prototype = {} or Rabbit.prototype = { newProperty: true }), we are changing the prototype reference for the constructor function itself, not the prototype object that existing instances reference.

This means that only new instances created after the change will have the new prototype object. Existing instances retain the link to the original prototype object that they were linked to when they were created.

4. What will be the output

```
const animal = {
 jumps: null,
};
const rabbit = {
 __proto__: animal,
 jumps: true,
};

alert(rabbit.jumps); // ? (1)

delete rabbit.jumps;

alert(rabbit.jumps); // ? (2)

delete animal.jumps;

alert(rabbit.jumps); // ? (3)
```