Prototypes in JavaScript

All objects in JavaScript are instances of **Object** like a father of a Nation. When an object is created that inherits properties and methods from **Object.prototype**. These properties and methods can be overridden (shadowed). When a constructor function (Blueprints) or a method (Blueprints or Classes method) is created in JavaScript, JavaScript engine adds a **prototype property** to the constructor function or inside a method. This prototype property is an object (called as prototype object) which has a constructor property by default and it points back to its containing function. We can access the function's prototype property using the below syntax:

For Constructor:

**ConstructorName.prototype**

For Method:

**methodName.prototype**

“Remember, a Constructor function and Classes are Blueprints in Javascript. So we can create an object from both a Constructor and Class in Javascript.”

A Constructor’s name begins with capital letter. Let’s consider a person constructor function as follows:

function Person(firstName, lastName){

this.firstName = firstName,

this.lastName = lastName,

this.fullName = function(){

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

}

}

console.log(Person.prototype);

**Output**:

constructor: ƒ Person(firstName, lastName)

\_\_proto\_\_: Object

As seen from the above output prototype property of the function is an object (prototype object) with two properties:

1. constructor property which points to Person function itself
2. \_\_proto\_\_ is the actual object that is used in the lookup chain. When we can create an object with **new** keyword the \_\_proto\_\_ is created as a property of that object.
3. We will discuss more while explaining inheritance in JavaScript

“In the prototype inheritance chain, the Object.prototype is on the topmost i.e. the Person constructor inherits properties from there nearest prototype if not found then finally search into Object.prototype, if not found, throws an exception.”

**Creating an object using the constructor function**

When an object is created in JavaScript, JavaScript engine adds a **\_\_proto\_\_** property to the newly created object which is called as dunder proto. The \_\_proto\_\_ points to the prototype object.

var person1 = new Person("Bablu", "Ahmed");

console.log(person1);

**Output:**

firstName: "Bablu"

lastName: "Ahmed"

fullName: ƒ ()

\_\_proto\_\_: Object

As shown in the above output, person1 object which is created using the Person constructor function has a dunder proto or \_\_proto\_\_ property which points to the prototype object of the constructor function.

As it can be seen from the output of Person.prototype and person1 , both person1's **\_\_proto\_\_** property and **Person.prototype** property are equal let's check if they point at the same location.

Person.prototype === person1.\_\_proto\_\_ //true

This shows that person1's dunder proto property and Person.prototype are pointing to the same object.

**Now, let's create another object person2 using the Person constructor function:**

var person2 = new Person("Fazlul", "Haque");

console.log(person2);

**Output**:

firstName: "Fazlul"

lastName: "Haque"

fullName: ƒ ()

\_\_proto\_\_: Object

Above console output shows that even person2's dunder proto property is equal to the **Person.prototype** property and they point to the same object

Person.prototype === person2.\_\_proto\_\_ //true

**Now, Let's verify if person1's dunder proto and person2's dunder proto:**

person1.\_\_proto\_\_ === person2.\_\_proto\_\_ //true

“From the above statement, we proved that the person1's and person2's dunder proto properties point to Person constructor function's **prototype object**”

Prototype object of the constructor function is shared among all the objects created using the constructor function.

**Prototype object**

A prototype object is an object, we can attach properties and methods to the prototype object. Thus, enabling all the objects created using the constructor function to share the properties and methods.

We know that a constructor function's prototype property has a property called dunder proto by default when created the constructor a new property can be added to the constructor function's prototype property using either the dot notation or square bracket notation as shown below:

//Dot notation

Person.prototype.name = "Sakib";

console.log(Person.prototype.name) //Output: Sakib

//Square bracket notation

Person.prototype["age"] = 26;

console.log(Person.prototype["age"]); //Output: 26

console.log(Person.prototype);

**Output**:

name: "Sakib"

age: 26

constructor: ƒ Person(firstName, lastName)

\_\_proto\_\_: Object

In the above output we can see name and age properties have been added to **Person.prototype** object

**For example, let’s create an empty constructor function as follows:**

function Person(){

}

**Add property name, age to the prototype property of the Person constructor function:**

Person.prototype.name = "Sakib" ;

Person.prototype.age = 26;

Person.prototype.theName = function(){

console.log(this.name); //Here, **this** is referring to the prototype object

}

**Now, create an object using the Person constructor function:**

var person1 = new Person();

**Access the name property using the Person object:**

console.log(person1.name) // Sakib

**Let's check if the person1 object has name property:**

console.log(person1);

**Output**:

\_\_proto\_\_: Object

As we can see that person1 object is empty and it does not have any property except it's dunder proto property. So how does the output of console.log(person1.name) was "Sakib"

*When we try to access a property of an object, the seach for the property begins directly on the object itself. If a property with a given names is found on the instance, then that value is returned; if the property is not found, then the search continues up the pointer to the prototype of the Object, and the prototype is searched for a property with the same name. If the property is found on the prototype, then that value is returned.*

So, when person1.name is called, JavaScript engine checks if the property exsit on the person object. In this case, name property was not on the person1's object. So, now JavaScript engine checks if the name property exists on the dunder proto property. In this cases, name property was there on the dunder proto property or the prototype of person's object. Hence, the output was returned "Sakib". If the dunder proto property of the person's object does not have the name property then dunder proto property of the dunder proto property of the person's object was searched and this process will continue till the dunder proto property is null. In this cases, output will be **undefined**.

**N.B,** when we create an object from a constructor we don’t have the property called prototype i.e. person1.prototype, instead of this we have \_\_proto\_\_ i.e. person1.\_\_proto\_\_

**Let's create another object person2 using the Person constructor function:**

var person2 = new Person();

**Access the name property using the person2 object:**

console.log(person2.name)// Output: Sakib

**Now, let's define a property name on the person1 object:**

person1.name = "Rakib" //Overridden

console.log(person1.name)//Output: Rakib

console.log(person2.name)//Output: Sakib

This happened because, when we define a property on the object itself, JavaScript engine takes the name property from the object itself and not from the objects prototype property i.e. person1 object's name property shadows the name property of the prototype object. Hence, person2 does not have name property, it looks up to the prototype to get the name property.

**Reference type in JavaScript**

Arrays are called reference type in JavaScript. Prototype object of the constructor function is shared among all the objects created using the constructor function. All properties on the prototype are shared among all the objects created using the constructor function. Properties that contain primitive values also tend to work well, as shown in the previous example, where it’s possible to hide the prototype property by assigning a property of the same name to the object. The real problem occurs when a prototype object contains a property of reference type. Consider the following example:

**Modifying the primitive type properties work well as shown below:**

person1.name = "Fahim"

console.log(perosn1.name); //Output: Fahim

console.log(person2.name); //Output: Sakib

Consider another example to display an issue with prototypes when the prototype object contains a property of reference type

Let’s create an empty constructor function:

function Person(){

}

**Add property name, age to the prototype property of the Person constructor function:**

Person.prototype.name = "Sakib" ;

Person.prototype.age = 26;

Person.prototype.friends = ['Bablu', 'Rakib'], // reference type in JavaScript

Person.prototype.yourName = function(){

console.log(this.name); //Sakib

}

**Now, create objects using the Person constructor function:**

var person1= new Person();

var person2 = new Person();

**Add a new element to the friends array:**

person1.friends.push("Rony");

console.log(person1.friends); // Output: "Bablu, Rakib, Rony"

console.log(person2.friends); // Output: "Bablu, Rakib, Rony"

Here, the Person.prototype object has a property called friends that contains an array of strings. Two objects of Person, person1 and person2 are created. person1 modifies friends property and adds another string in the array. Because the friends array exists on Person.prototype, not on person1, the changes made in the friends property by person1. So we can see that the object2. The friends array is also reflected. If the intention is to have an array shared by all instances, then this outcome is okay. Typically, though, instances want to have their own copies of all properties.

**Combine Constructor/Prototype**

To solve the problems with the prototype and the problems with the constructor, we can combine both the constructor and function.

**Problem with constructor**: Every object has a copy of the properties of the constructor but the problem is when a method is defined in the constructor that is not nesassary for every instances of the constructor because of wastage of memory.

**Problem with the prototype**: Modifying a property using one object reflects the other object also.

To solve above both problems, we can define all the object-specific properties inside the constructor and all shared properties and methods inside the prototype as shown below:

**Define the object-specific properties inside the constructor:**

function Person(name, age){

this.name = name,

this.age = age,

this.friends = ["Bablu", "Rakib"]

}

**Define the shared properties and methods using the prototype:**

Person.prototype.yourName = function(){

console.log(this.name);

}

**Create two objects using the Person constructor function:**

var person1 = new Person("Sakib", "Khan");

var person2 = new Person("Riaz", "Ahmed");

**Lets check if person1 and person2 point to the same instance of the yourName function:**

console.log(person1. yourName === person2. yourName) // true

**Let's modify friends property and check:**

person1.friends.push("Rony");

console.log(person1.friends) // Output: "Bablu, Rakib, Rony"

console.log(person2.frinds) //Output: "Bablu,Rakib"

friends property of person2 did not change on changing the friends property of person1