Prototypal Inheritance in Function Objects

This lesson teaches us the concept of prototypal inheritance in function objects as well as property shadowing.

WE'LL COVER THE FOLLOWING

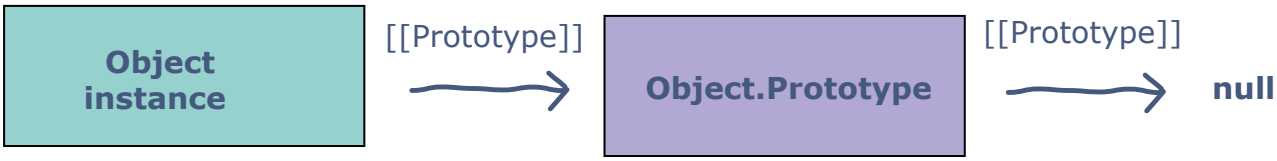
Prototypal inheritance from `Object`

How does prototypal inheritance/prototype chaining work when constructor functions are used to create objects? In a previous chapter, we discussed the **Prototype** object of *constructor functions*.

It was concluded that all properties (including methods) added to the prototype of the constructor function could be shared by the object instances created from it. Let's dive into the details now.

As mentioned earlier, every constructor function initialized has a *prototype property* assigned to it known as the **Prototype Object**. It contains the properties `constructor` and `__proto__` that sets the object's `[[Prototype]]` property. Its `[[Prototype]]` is set to `null`. All object instances created will have their `[[Prototype]]` reference the prototype object. This was seen in a previous chapter through the following images:

As seen in the diagram above, an object instance created from a constructor function will have its `[[Prototype]]` pointing to the `[[Prototype]]` object. This can be seen as the `constructor` in the object instance points to the `constructor` in the *prototype object* of the *constructor* function `EmployeeConstructor`.



Prototype Chain When Using Constructor Function

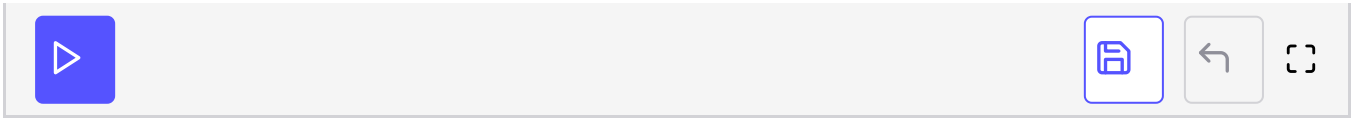
So it is established that all object instances created will have the *prototype object* as their prototype, and will inherit its properties through **prototypal inheritance**. So whenever a property is accessed in an object, it will first be searched for within the object; if it is not found, it'll get searched for and taken from the prototype object.

Example

Let's implement the concept above through an example:

```
//using constructor function to create Employee
function Employee(name,age,sex){
  this.name = name
  this.age = age
  this.sex = sex
```

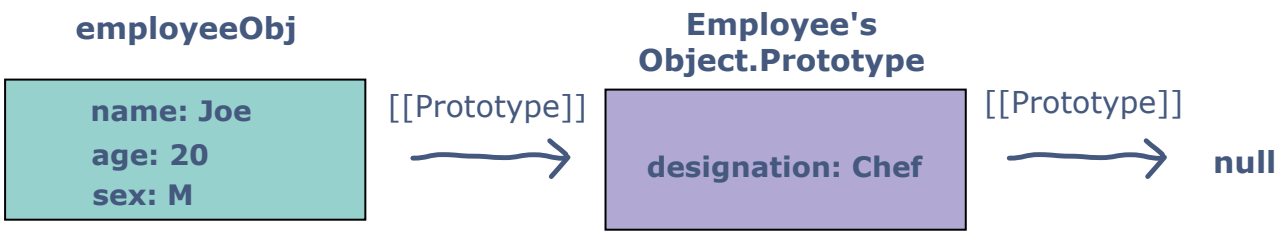
```
}
//The prototype of employeeObj will be Employee's Object prototype
var employeeObj = new Employee('Joe',20,'M')

//This will show Employee's Object.prototype as it is set as employeeObj's [[Prototype]]
//The constructor and __proto__ properties are only visible
//since no other properties are defined on it the console will display an empty object
console.log(employeeObj.__proto__)
//name,age and sex are all properties of employeeObj
console.log("Name:",employeeObj.name)
console.log("Age:",employeeObj.age)
console.log("Sex:",employeeObj.sex)
//adding property to prototype of Employee
Employee.prototype.designation = "Chef"
//employeeObj.__proto__ is equal to Employee's Object prototype which will now contain "desig
//so "designation:chef" will be displayed
console.log(employeeObj.__proto__)
//accessing designation from employeeObj
//the prototype chain will start to be traversed
//first it will search employeeObj upon not finding it
//it will search employeeObj.[[Prototype]] which is the Employee Object prototype
//here it will find "designation" and its value which it will then return
console.log("Designation:",employeeObj.designation)
```



Explanation

As seen from the code above:

- At the start, the **Prototype Object** of `Employee` is empty as seen from **line 10**.
- The `employeeObj` object created will have its `[[Prototype]]` set to the **Prototype Object** of `Employee`.
- The property `designation` is added to the `Employee` prototype.
- Now the **Prototype Object** of `Employee` will contain the `designation` property in it as seen from **line 18**.
- The prototype chain that is built will be as follows:



- When `designation` is accessed from `employeeObj`, the chain will be

traversed.

- The property won't be found in `employeeObj`.
- Next, `employeeObj.[[Prototype]]` will be traversed where the property will be found.
- The value of `designation` will then be taken from here and displayed.

Property Shadowing

The working of the prototype chain was discussed above. The whole chain is traversed and the property required is returned if it is found in the chain. However, what happens if a property exists in two places on the chain?

Example

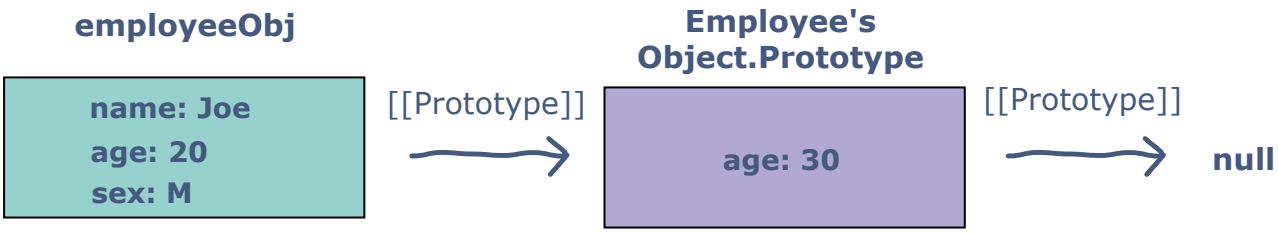
Let's find out through the example given below:

```
//using constructor function to create Employee
function Employee(name,age,sex){
  this.name = name
  this.age = age
  this.sex = sex
}
//The prototype of employeeObj will be Employee's Object prototype
var employeeObj = new Employee('Joe',20,'M')
//This will show Employee's Object.prototype which is null as employeeObj's [[Prototype]] pro
console.log(employeeObj.__proto__)
//name,age and sex are all properties of employeeObj
console.log("Name:",employeeObj.name)
console.log("Sex:",employeeObj.sex)
//adding age property to prototype of Employee
Employee.prototype.age = 30
//employeeObj.__proto__ is equal to Employee's Object prototype
//It will contain "age:30" now
console.log(employeeObj.__proto__)
//accessing "age" from employee1
//the prototype chain will be start to be traversed
//first it will search employeeObj
//the property "age" will be found which will be returned
//it will not search employeeObj.[[Prototype]]
//hence the age displayed will be 20 instead of 30
console.log("Age:",employeeObj.age)
```

Explanation

As seen from the code above:

- The `Employee` function contains the property `age`.
- The property `age` is also added to the `prototype` of `Employee`.
- The prototype chain now looks like this:



- When traversal begins, `employeeObj` is searched first where `age` is found and returned.
- Since `age` is found in `employeeObj`, the `age` property in `Employee.prototype` is not visited. This is known as **property shadowing**.

Now that we have discussed prototypal inheritance from a *constructor function*, let's discuss how to implement class-based inheritance using the *constructor functions* in the next lesson.