**03 Prototypes**

**1) Inheritance**:

Inheritance is one of the core concepts of object-oriented programming that enables an object to take on the properties and methods of another object. And this makes it easy to reuse code in different parts of an application.

**Example**:

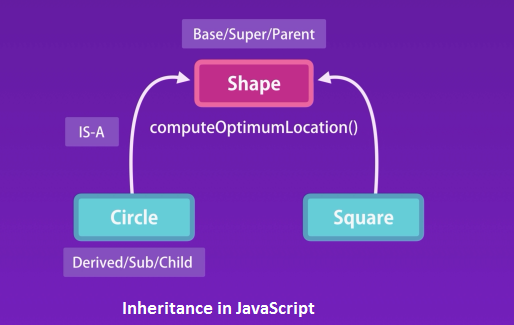
Suppose we have a "Circle" class and inside it we have "computeOptimumLocation()" method. Now in our application we have another class called "Square" and it also need the method same method "computeOptimumLocation ()".

Imagine the implementation of this method is exactly the same across these two different classes; we don’t want to repeat this implementation. Because if there is a bug in this implementation, we have to fix it in multiple places. Or if we want to change the implementation and make it better, again, we have to repeat this in different places.

For resolve this type of problem inheritance concept comes to the rescue.

We can define a new class called "Shape", put the "computeOptimumLocation ()" in this class. Then "Circle" and "Square" inherit this method, from the "Shape" class.

**Diagram**:



**Classical vs. Prototypical Inheritance in JavaScript**:

In the above we discuss the classical definition of inheritance. But in JavaScript, we don’t have classes, we only have objects. For these reason prototypical inheritances comes in the picture. So essentially we have two types of inheritance, classical and prototypical.

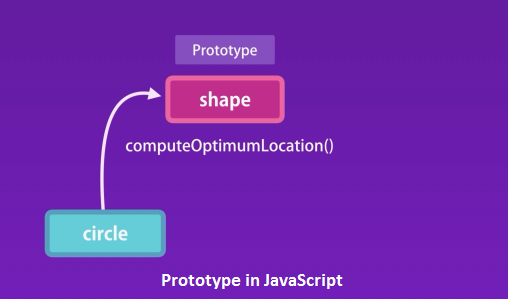
**2) Prototypes and Prototypical Inheritance**:

In JavaScript we don’t have classes. We only have objects. In JavaScript we have to implements inheritance only using objects.

Here we have a "Shape" object. We can define another object name "Circle" and add all the common behavior of "Shape" object like in "Circle" Object. Suppose in "Shape" object we have "computeOptimumLocation ()" method and we want to add this in "Circle" object. For this, somehow we have to create a link between the two objects. Now we refer to the "Shape" object as the prototype of the "Circle" object.

So prototype is essential a parent of another object. Every object in JavaScript (Except only a single object) has a prototype or parent. And it inherits all the members define in its prototype.

**Diagram**:



**Example**:

In Firefox developer tool in the console define a new object named x.

let x = {}

If we print x we will see a property called "Object { }". This property is duplicated and we don’t directly access it. This only available here to only help for troubleshoot problems.

If we expand this we will see the prototype or parent for the x object

{}

<prototype>: {…}

\_\_defineGetter\_\_: function \_\_defineGetter\_\_()

\_\_defineSetter\_\_: function \_\_defineSetter\_\_()​​

\_\_lookupGetter\_\_: function \_\_lookupGetter\_\_()​​

\_\_lookupSetter\_\_: function \_\_lookupSetter\_\_()​​

…………..

**Diagram**:



We have x object in memory and x has a link to another object which is its prototype. For discussion lets call this object objectBase (It’s not an official name).

So every object that we create in JavaScript directly or indirectly inherits from object base. ObjectBase is the root of all objects in JavaScript and it doesn’t have a prototype or parent.

**Diagram**:

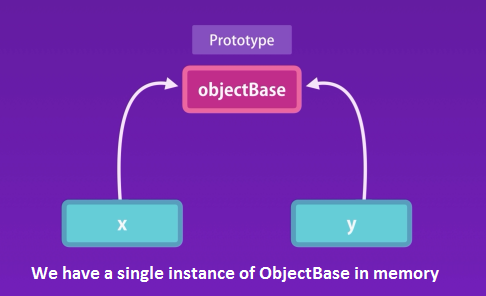


Here we see that ObjectBase doesn’t have a prototype or parent.

**We have a single instance of ObjectBase in memory**:

All of JavaScript object reference to ObjectBase and it has a single instance in memory.

**Diagram**:



**Example**:

let x = {};

let y = {};

let c = Object.getPrototypeOf(x) === Object.getPrototypeOf(y);

console.log(c); *//true*

console.log(x.\_\_proto === y.\_\_proto); *//true*

When we access a property or a method on an object JavaScript engine first looks for that property or method on that object itself. If it cannot find it, then it looks at the prototype for that object. This is prototypical inheritance.

When accessing a prototype or a method on an object JavaScript engine works up the prototype chain to find the target member.

**3) Multi-level Inheritance**:

JavaScript support multilevel inheritances. In console if we declare an array and we see the property of the array we will see the following property.

let myArray = [];

myArray

[]​

length: 0​

<prototype>: []​​ **//prototype property of the array**

concat: function concat()​​

constructor: function Array()​​

copyWithin: function copyWithin()​​

entries: function entries()​​

every: function every()​​

fill: function fill()​​

filter: function filter()​​

find: function find()​​

findIndex: function findIndex()​​

flat: function flat()​​

flatMap: function flatMap()​​

forEach: function forEach()​​

includes: function includes()​​

indexOf: function indexOf()​​

join: function join()​​

keys: function keys()

​​lastIndexOf: function lastIndexOf()

​​length: 0​​

map: function map()

​​pop: function pop()

​​push: function push()

​​reduce: function reduce()

​​reduceRight: function reduceRight()

​​reverse: function reverse()

​​shift: function shift()

​​slice: function slice()

​​some: function some()

​​sort: function sort()

​​splice: function splice()

​​toLocaleString: function toLocaleString()

​​toSource: function toSource()

​​toString: function toString()

​​unshift: function unshift()

​​values: function values()

​​Symbol(Symbol.iterator): function values()

​<prototype>: {…} **// another proto property which is object base**

​​​\_\_defineGetter\_\_: function \_\_defineGetter\_\_()

​​​\_\_defineSetter\_\_: function \_\_defineSetter\_\_()

​​​\_\_lookupGetter\_\_: function \_\_lookupGetter\_\_()

​​​\_\_lookupSetter\_\_: function \_\_lookupSetter\_\_()

​​​constructor: function Object()

​​​hasOwnProperty: function hasOwnProperty()

​​​isPrototypeOf: function isPrototypeOf()

​​​propertyIsEnumerable: function propertyIsEnumerable()

​​​toLocaleString: function toLocaleString()

​​​toSource: function toSource()

​​​toString: function toString()

​​​valueOf: function valueOf()

​Here we have a prototype property of the array. All this methods are define in this array objects which

is the prototype or parent for all arrays in JavaScript. So in memory we have

**myArray ------(derive from)------> arrayBase**

After that we have another proto property which is object base. This is the root object in JavaScript. In memory

**myArray ------(derive from)------> arrayBase ------(derive from)------> objectBase**

This process is called multilevel inheritances.

**Multilevel inheritance and custom constructor**:

Suppose we have a constructor named "Circle" and using it we create a "circle" object.

function Circle(radious){

*this*.radious = radious;

*this*.draw = function(){

console.log("draw");

};

}

const circle = new Circle(10);

In Console

circle

Circle {radious: 10, draw: ƒ}

draw: ƒ ()

radious: 10

\_\_proto\_\_: //protype for all circle object

constructor: ƒ Circle(radious)

\_\_proto\_\_: //Object Base

constructor: ƒ Object()

hasOwnProperty: ƒ hasOwnProperty()

isPrototypeOf: ƒ isPrototypeOf()

propertyIsEnumerable: ƒ propertyIsEnumerable()

toLocaleString: ƒ toLocaleString()

toString: ƒ toString()

valueOf: ƒ valueOf()

\_\_defineGetter\_\_: ƒ \_\_defineGetter\_\_()

\_\_defineSetter\_\_: ƒ \_\_defineSetter\_\_()

\_\_lookupGetter\_\_: ƒ \_\_lookupGetter\_\_()

\_\_lookupSetter\_\_: ƒ \_\_lookupSetter\_\_()

get \_\_proto\_\_: ƒ \_\_proto\_\_()

set \_\_proto\_\_: ƒ \_\_proto\_\_()

Here for circle object we have a prototype. This prototype is the prototype for all circle objects that we create using our Circle constructor. Every time we call the circle constructor "new Circle(10)" to create a new circle object this constructor will create a new object and set it’s prototype to circle base.

In other words object created by a giving constructor will have the same prototype. All circle objects created by the Circle constructor will have the same prototype, and similarly all arrays object created by the array constructor have same prototype.

So in memory

**circle ------(derive from)------> circleBase ------(derive from)------> objectBase**

**4) Property Descriptors**:

Every object property is more than just a name and value pair. Each property is having property descriptor that helps to see all the attributes of that property.

Suppose we have an object an it has a property call “name”. If we print the object, we will see the following output.

**Example**:

let person = { name: "Ruhul" };

console.log(person);

*/\**

*{name: "Ruhul"}*

*name: "Ruhul"*

*\_\_proto\_\_:*

*constructor: ƒ Object()*

*hasOwnProperty: ƒ hasOwnProperty()*

*isPrototypeOf: ƒ isPrototypeOf()*

*propertyIsEnumerable: ƒ propertyIsEnumerable()*

*toLocaleString: ƒ toLocaleString()*

*toString: ƒ toString()*

*valueOf: ƒ valueOf()*

*\_\_defineGetter\_\_: ƒ \_\_defineGetter\_\_()*

*\_\_defineSetter\_\_: ƒ \_\_defineSetter\_\_()*

*\_\_lookupGetter\_\_: ƒ \_\_lookupGetter\_\_()*

*\_\_lookupSetter\_\_: ƒ \_\_lookupSetter\_\_()*

*get \_\_proto\_\_: ƒ \_\_proto\_\_()*

*set \_\_proto\_\_: ƒ \_\_proto\_\_()*

*\*/*

If we iterate over the members of the object, we are not going to see the members present in the prototype. We will see only “name”. None of the property and method objectBase are available here.

**Example**:

let person = { name: "Ruhul" };

for (let key in person) {

console.log(key); *//name*

}

console.log(Object.keys(person)); *//["name"]*

We cannot iterate over all these properties and methods define in object base because in JavaScript our properties have attributes attached to them. Sometimes these attributes prevent a property from being enumerated.

**Example**:

let person = { name: "Ruhul" };

*//get the prototype of person*

let objecBase = Object.getPrototypeOf(person);

*//the attributes attach to the toString() method*

let descriptor = Object.getOwnPropertyDescriptor(objecBase, "toString");

console.log(descriptor);

*/\**

*{value: ƒ, writable: true, enumerable: false, configurable: true}*

*configurable: true*

*enumerable: false*

*value: ƒ toString()*

*writable: true*

*\_\_proto\_\_: Object*

*\*/*

"configurable: true" means we can delete this member if we want to.

"enumerable: false" means we can’t iterate over this property.

"writable: true" means we can overwrite this method. We can change its implementation.

Now when we create our own objects, we can set this attribute for our property.

**Example**:

let person = { name: "Ruhul" };

Object.defineProperty(person, "name", {

writable: false,

enumerable: true

});

console.log(person.name); *//Ruhul*

person.name = "Reza"; *//name not set because "writable: false"*

console.log(person.name); *//Ruhul*

console.log(Object.keys(person)); *//["name"]*

Now we set "writable: true" and "enumerable: false".

**Example**:

let person = { name: "Ruhul" };

Object.defineProperty(person, "name", {

writable: true,

enumerable: false

});

console.log(person.name); *//Ruhul*

person.name = "Reza"; *//name set because "writable: true"*

console.log(person.name); *//Reza*

console.log(Object.keys(person)); *//[]*

If we set "configurable: false" we cannot delete it

**Example**:

let person = { name: "Ruhul"};

Object.defineProperty(person, "name", {

writable: true,

enumerable: false,

configurable: false

});

delete person.name; *//not delete because "configurable: false"*

console.log(person.name); *//Ruhul*

If we set "configurable: true" then we can delete it

**Example**:

let person = { name: "Ruhul"};

Object.defineProperty(person, "name", {

writable: true,

enumerable: false,

configurable: true

});

delete person.name; *//delete because "configurable: true"*

console.log(person.name); *//undefined*

**5) Constructor Prototypes**:

Every object in JavaScript except the rootObject has a prototype or a parent. The proper way to get prototype of an object is

"Object.getPrototypeOf(myObject)" //parent of myObject

Or

"person.\_\_proto\_\_"

**Example**:

function Circle(radius) {

*this*.radius;

}

const circle = new Circle(20);

console.log(Object.getPrototypeOf(circle));

console.log(circle.\_\_proto\_\_); *//parent of circle => Circle*

console.log(Circle.prototype);

*/\**

*{constructor: ƒ}*

*constructor: ƒ Circle(radius)*

*\_\_proto\_\_: Object*

*{constructor: ƒ}*

*constructor: ƒ Circle(radius)*

*\_\_proto\_\_: Object*

*{constructor: ƒ}*

*constructor: ƒ Circle(radius)*

*\_\_proto\_\_: Object*

*\*/*

We know that the constructor also has a prototype property. Here we have the Circle constructor function and we know that in JavaScript functions are object so they have property’s and methods.

In circle object we have a property call prototype ("Circle.prototype"). This is the object that will be used as the parent for object created by the circle constructor.

**ObjectBase**:

let obj = {};

console.log(obj.\_\_proto\_\_);

*/\**

*{constructor: ƒ, \_\_defineGetter\_\_: ƒ, \_\_defineSetter\_\_: ƒ, hasOwnProperty: ƒ, \_\_lookupGetter\_\_: ƒ, …}*

*constructor: ƒ Object()*

*hasOwnProperty: ƒ hasOwnProperty()*

*isPrototypeOf: ƒ isPrototypeOf()*

*propertyIsEnumerable: ƒ propertyIsEnumerable()*

*toLocaleString: ƒ toLocaleString()*

*toString: ƒ toString()*

*valueOf: ƒ valueOf()*

*\_\_defineGetter\_\_: ƒ \_\_defineGetter\_\_()*

*\_\_defineSetter\_\_: ƒ \_\_defineSetter\_\_()*

*\_\_lookupGetter\_\_: ƒ \_\_lookupGetter\_\_()*

*\_\_lookupSetter\_\_: ƒ \_\_lookupSetter\_\_()*

*get \_\_proto\_\_: ƒ \_\_proto\_\_()*

*set \_\_proto\_\_: ƒ \_\_proto\_\_()*

*\*/*

This object is created by object constructor function. All object created by Object have the property.

**Example**:

console.log(Object.prototype);

*/\**

*{constructor: ƒ, \_\_defineGetter\_\_: ƒ, \_\_defineSetter\_\_: ƒ, hasOwnProperty: ƒ, \_\_lookupGetter\_\_: ƒ, …}*

*constructor: ƒ Object()*

*hasOwnProperty: ƒ hasOwnProperty()*

*isPrototypeOf: ƒ isPrototypeOf()*

*propertyIsEnumerable: ƒ propertyIsEnumerable()*

*toLocaleString: ƒ toLocaleString()*

*toString: ƒ toString()*

*valueOf: ƒ valueOf()*

*\_\_defineGetter\_\_: ƒ \_\_defineGetter\_\_()*

*\_\_defineSetter\_\_: ƒ \_\_defineSetter\_\_()*

*\_\_lookupGetter\_\_: ƒ \_\_lookupGetter\_\_()*

*\_\_lookupSetter\_\_: ƒ \_\_lookupSetter\_\_()*

*get \_\_proto\_\_: ƒ \_\_proto\_\_()*

*set \_\_proto\_\_: ƒ \_\_proto\_\_()*

*\*/*

The Array base constructor have some property and when we create any array this property is available in the array object as ArrayBase.

**Example**:

let array = [];

*//both are same*

console.log(Array.prototype);

console.log(array.\_\_proto\_\_);

*/\**

*[constructor: ƒ, concat: ƒ, copyWithin: ƒ, fill: ƒ, find: ƒ, …]*

*[constructor: ƒ, concat: ƒ, copyWithin: ƒ, fill: ƒ, find: ƒ, …]*

*\*/*

**6) Prototype vs. Instance Members**:

When we are working with JavaScript in real world application our object might have a large number of methods. If we have a large number of these object then it wastes a lot of memory by keeping copies of all this method.

To solve this problem, we can use prototypical inheritance. In JavaScript when we access a property or a method on an object, JavaScript engine first looks at the objects itself. If it cannot find that property or method, it will look the property of that object.

So, the commonly used method we remove from the object and put it in its prototype. Then we have a single instance of this prototype in the memory which will call circleBase. In this case we have a single instance of the draw() method.

**Example**:

function Circle(radius) {

*this*.radius = radius;

*this*.draw = function() {

console.log("draw");

};

}

const circle1 = new Circle(10);

const circle2 = new Circle(20);

*/\**

*circle1=>*

*Circle {radius: 10, draw: ƒ}*

*draw: ƒ ()*

*radius: 10*

*\_\_proto\_\_: Object*

*circle2=>*

*Circle {radius: 20, draw: ƒ}*

*draw: ƒ ()*

*radius: 20*

*\_\_proto\_\_: Object*

*\*/*

**Create single instance of draw method**:

function Circle(radius) {

*this*.radius = radius;

}

Circle.prototype.draw = function() {

console.log("draw");

};

const circle1 = new Circle(10);

const circle2 = new Circle(20);

*/\**

*circle1=>*

*Circle {radius: 10}*

*radius: 10*

*\_\_proto\_\_: Object*

*circle2=>*

*Circle {radius: 20}*

*radius: 20*

*\_\_proto\_\_: Object*

*\*/*

Now in the console we see that only radius property. The “draw()” method is on the prototype of circleBase object.

*/\**

*circle1*

*Circle {radius: 10}*

*radius: 10*

*\_\_proto\_\_:*

*draw: ƒ ()*

*constructor: ƒ Circle(radius)*

*\_\_proto\_\_: Object*

*\*/*

Now for prototypical inheritance, we can still access it. So we can call " circle.draw()".

console.log(circle1.draw()); *//draw*

console.log(circle2.draw()); *//draw*

We have two kinds of properties and methods in JavaScript.

1. Instance members
2. Prototype member

**Example**:

function Circle(radius) {

*//Instance members*

*this*.radius = radius;

}

*//Prototype member*

Circle.prototype.draw = function() {

console.log("draw");

};

**Overwrite method**:

We can overwrite the implementation of a method in the prototype of our Circle object.

**Example**:

function Circle(radius) {

*//Instance members*

*this*.radius = radius;

}

*//Prototype member*

Circle.prototype.draw = function() {

console.log("draw");

};

*//overwrite the implementation of toString() method*

Circle.prototype.toString = function() {

return "Circle radius: " + *this*.radius;

};

const circle1 = new Circle(10);

console.log(circle1.toString()); *//Circle radius: 10*

In both Prototype and Instance members we can reference other members.

**Example**:

function Circle(radius) {

*this*.radius = radius;

*this*.move = function() {

console.log("move");

};

}

Circle.prototype.draw = function() {

*this*.move();

console.log("draw");

};

const circle1 = new Circle(10);

console.log(circle1.draw());

*/\**

*move*

*draw*

*\*/*

Now in an instance method we can reference a prototype member.

**Example**:

function Circle(radius) {

*this*.radius = radius;

*this*.move = function() {

*this*.draw();

console.log("move");

};

}

Circle.prototype.draw = function() {

console.log("draw");

};

const circle1 = new Circle(10);

console.log(circle1.move());

*/\**

*draw*

*move*

*\*/*

**7) Iterating Instance and Prototype Members**:

**8) Avoid Extending the Built-in Objects**:

03 Prototypes