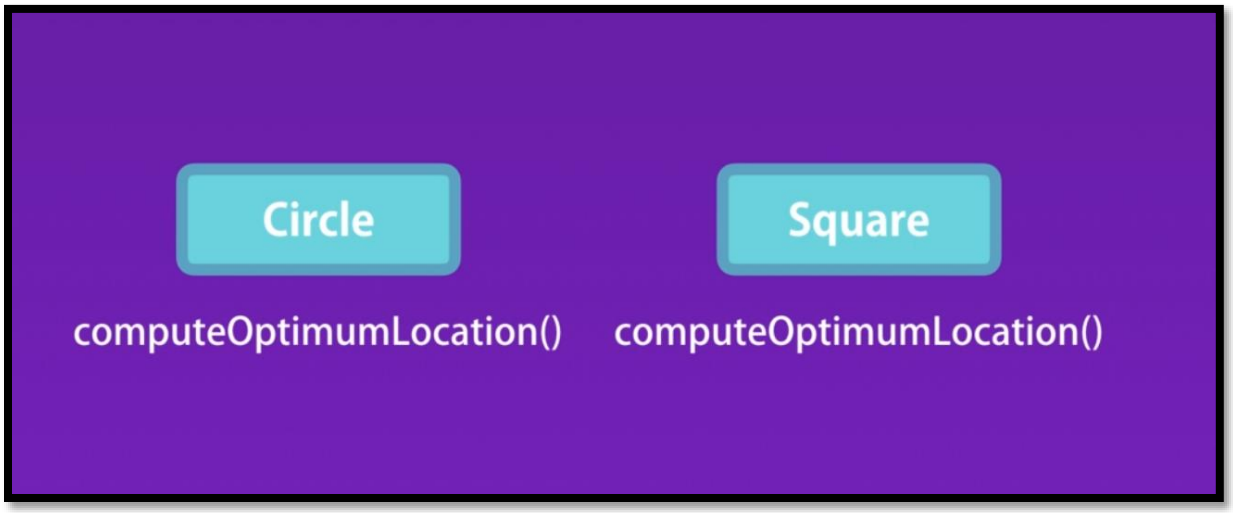
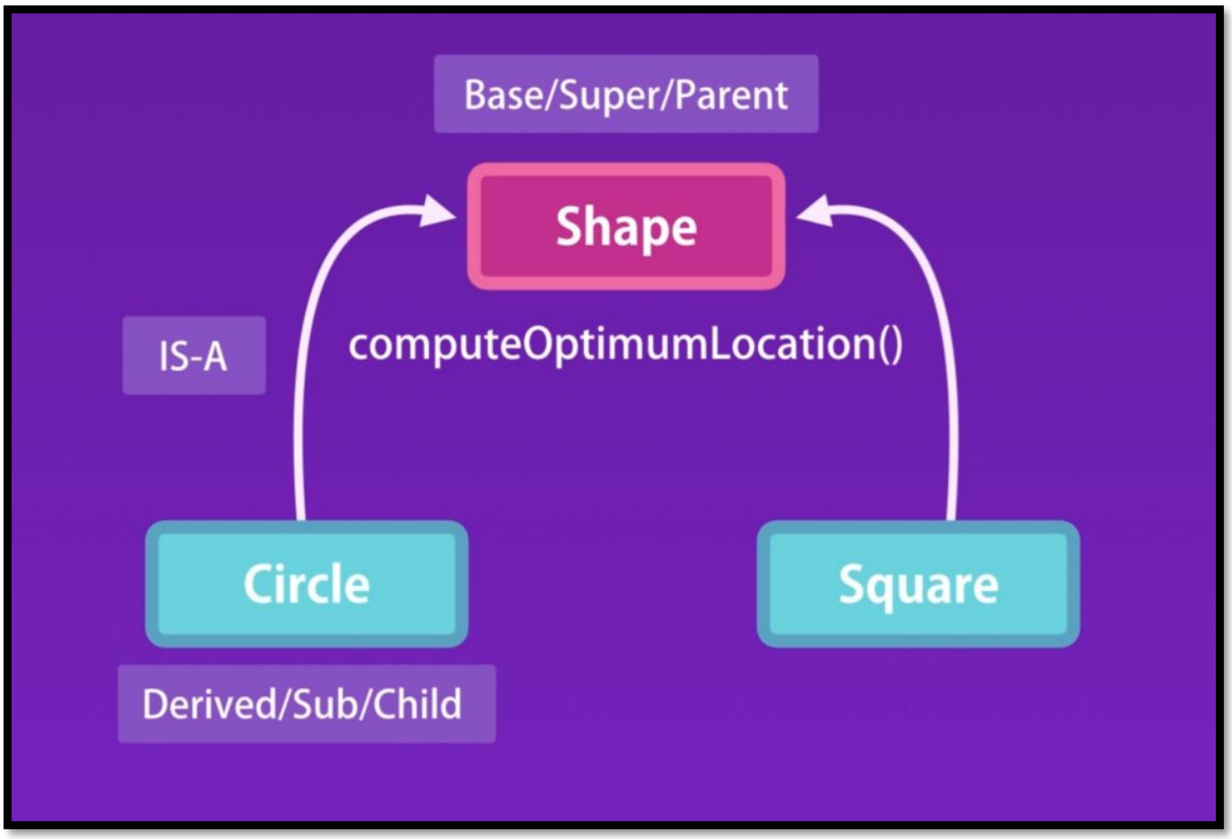
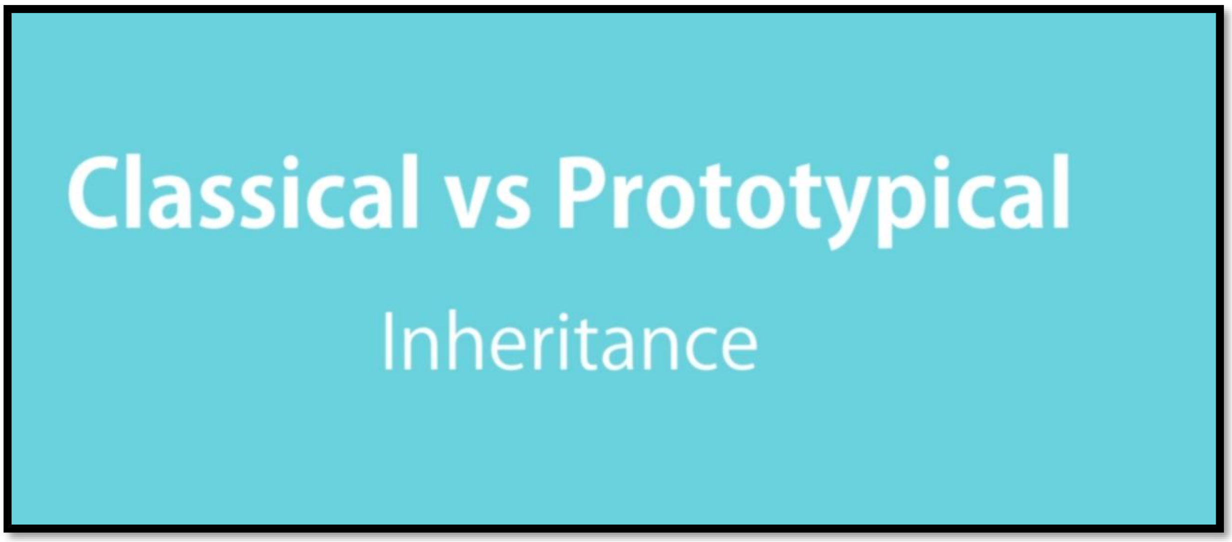
Inheritance

It is the core concept of Object-Oriented Programming that enables an object to take on the properties and methods of another object.

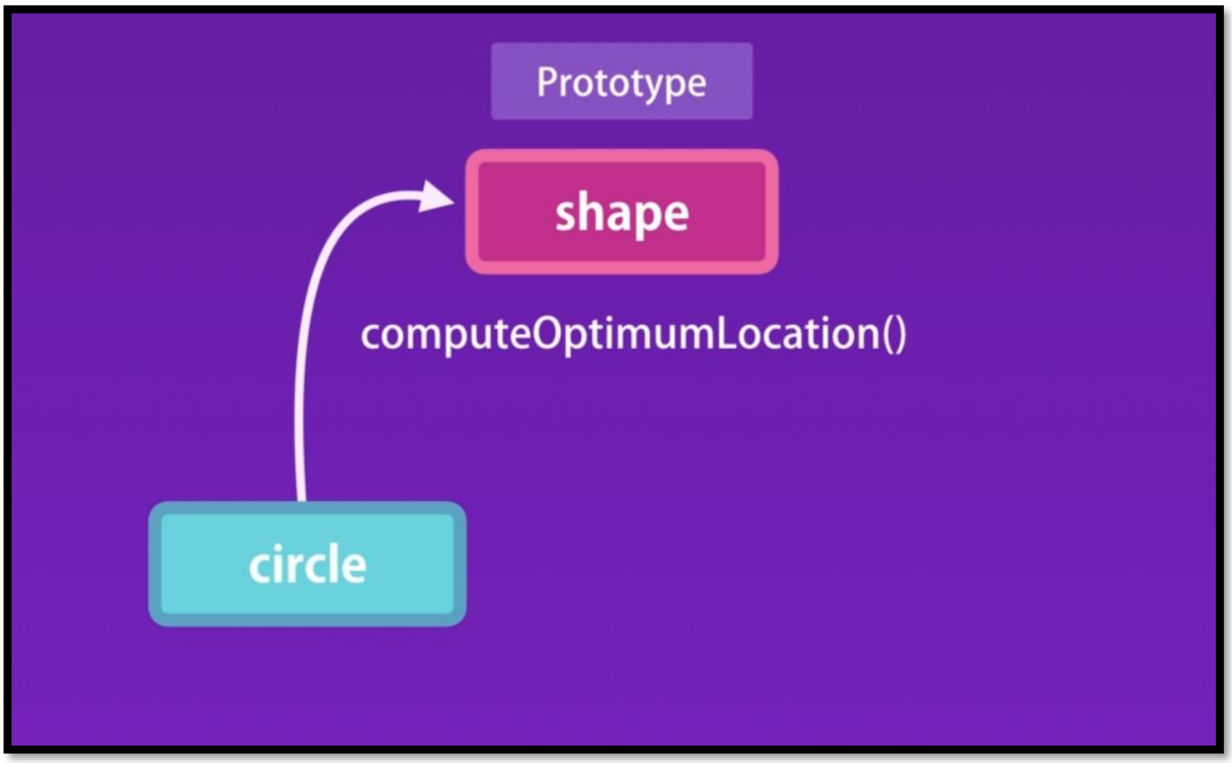


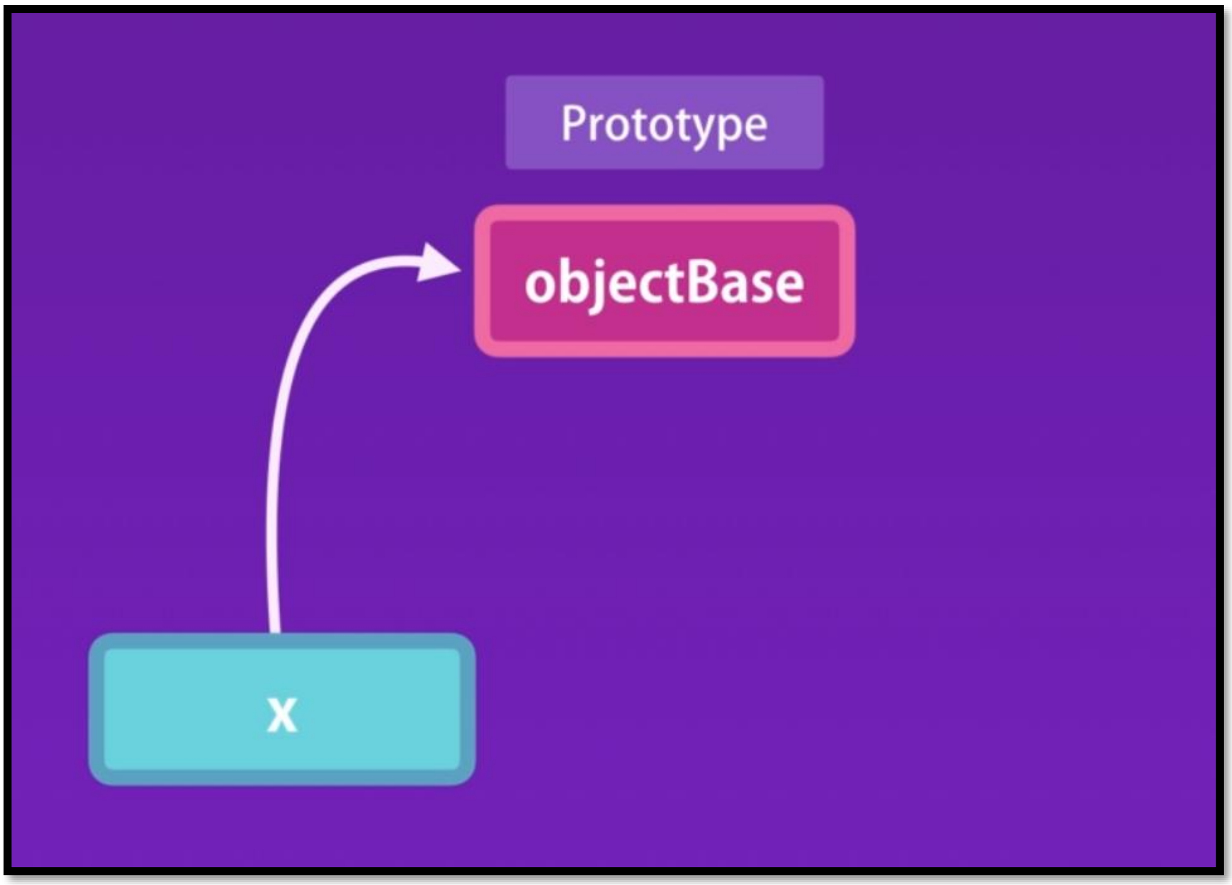


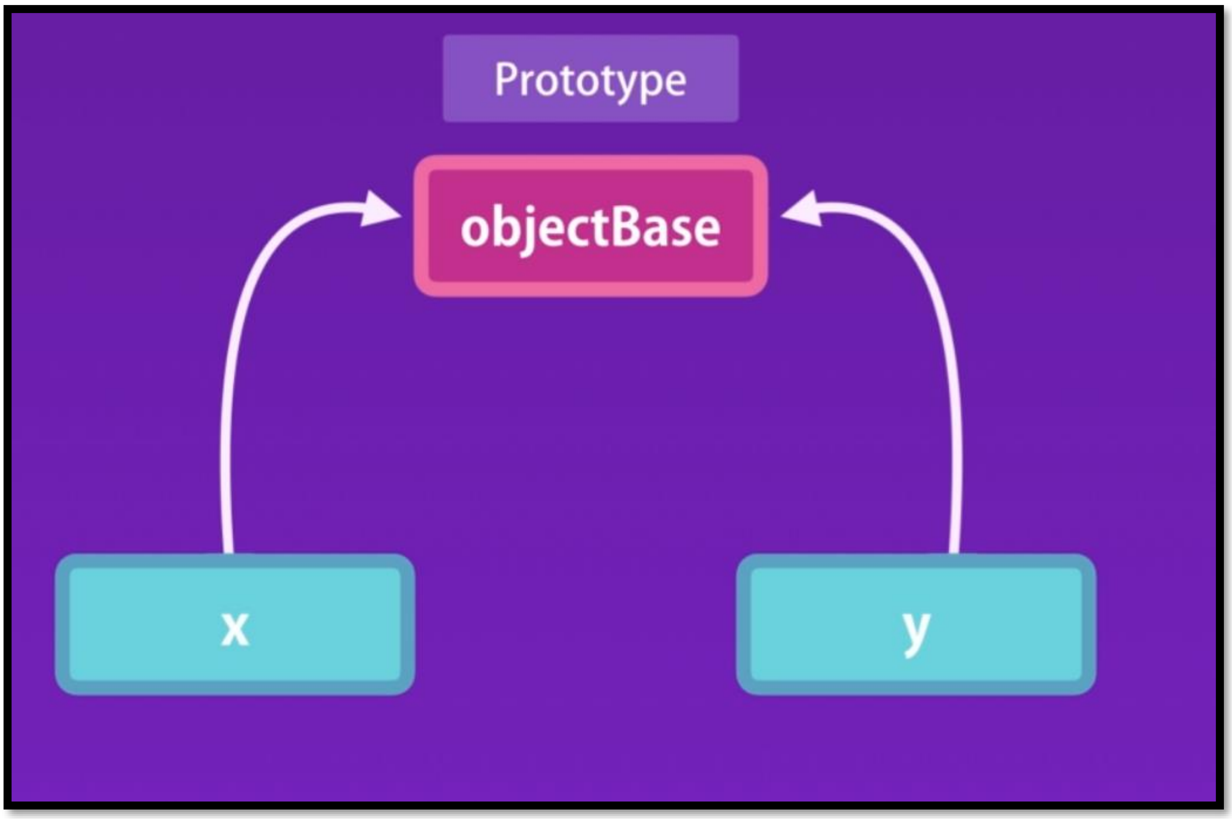


Classical vs Prototypical

Inheritance




Prototype
objectBase
x


Prototype
objectBase
x
y

```
let x = {};
let y = {};
let isSamePrototype = Object.getPrototypeOf(x) ===
Object.getPrototypeOf(y);
console.log(isSamePrototype); //true
let z = x.toString();
```

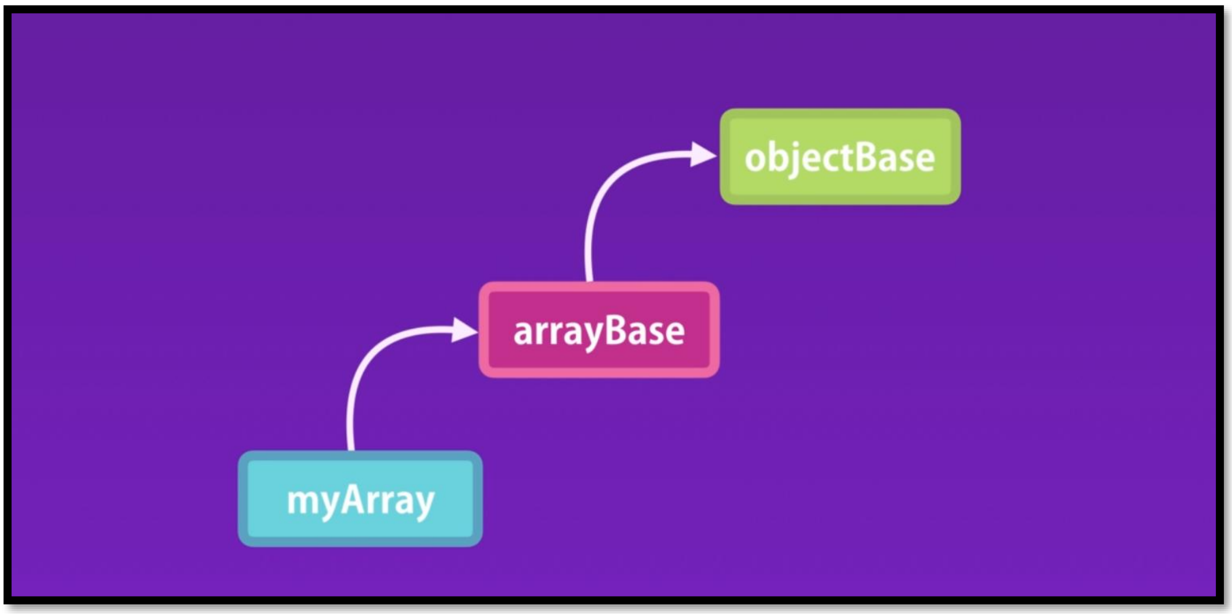
A constructor call makes an object linked to its own prototype.

Here the property is first checked in the object itself by the Javascript engine. If it does not find in the object then it checks its prototype. This behaviour is called prototypical inheritance.

Dunder Prototype

```
x.__proto__
```

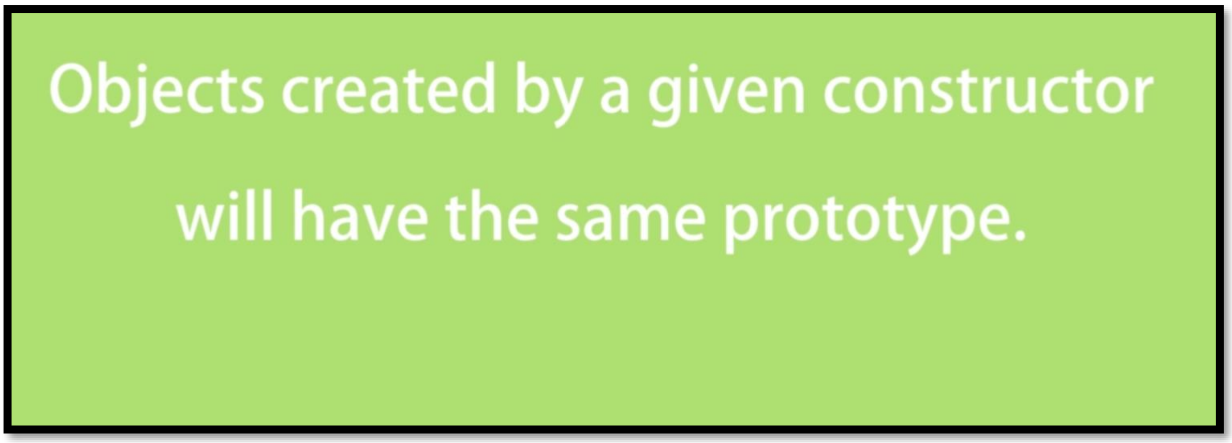
Multilevel Prototypical Inheritance



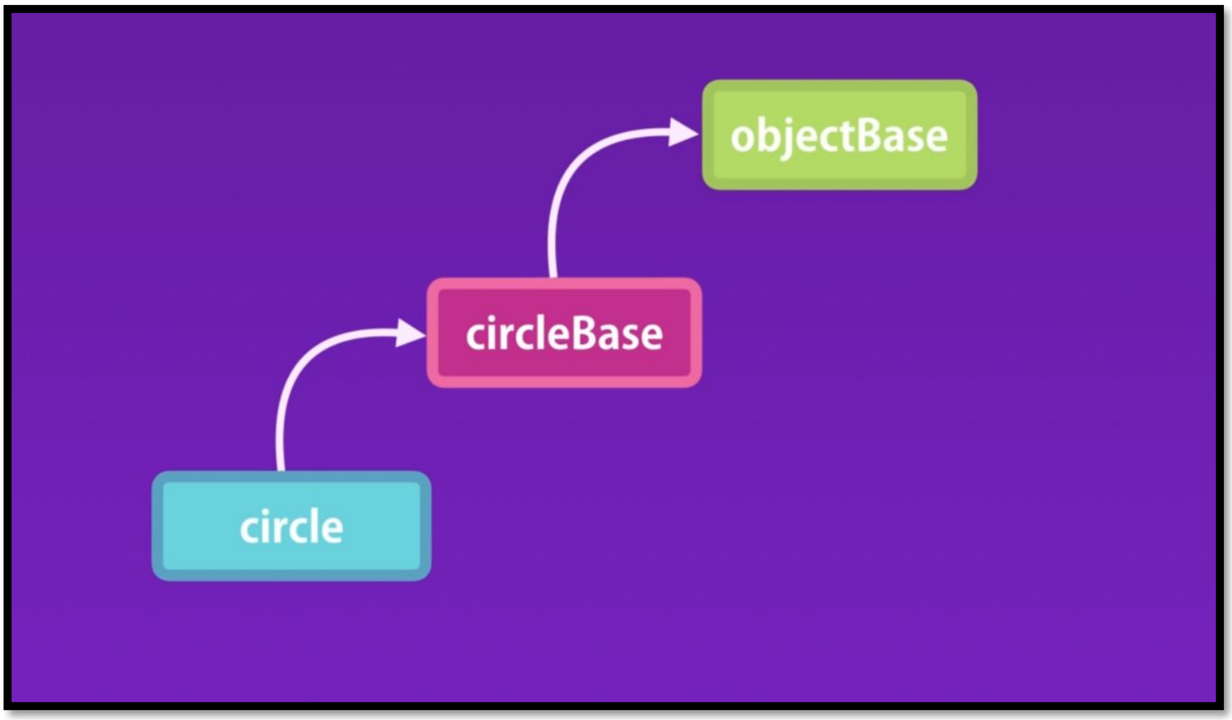
```
function Circle(radius)
{
    this.radius = radius;
    this.draw = function()
    {
        console.log('draw');
    }
}

const myCircle = new Circle(1);

myCircle -> circleBase -> objBase
```



Objects created by a given constructor will have the same prototype.



Property Descriptor

```
let person = { name: 'Anubhav Gupta'};
let objBase = Object.getPrototypeOf(person);
let propDesc = Object.getOwnPropertyDescriptor(objBase, 'toString');
console.log(propDesc);

//{ value: [Function: toString], writable: true, enumerable: false,
configurable: true }
```

configurable is true which means we can delete this member if we want to.
enumerable is false which means we can not iterate over.
writable is true which means we can overwrite this method if we want to.

```
let person = { name: 'Anubhav Gupta'};

Object.defineProperty(person, 'name',
{
    writable: false,
    enumerable: false,
    configurable: false
});

person.name = 'John';
console.log(person);// { name: 'Anubhav Gupta' }

delete person['name'];
console.log(person);// { name: 'Anubhav Gupta' }

//Object.keys can not be enumerated
```

Constructor Prototypes

Constructors also have a prototype property.

```
function Circle(radius)
{
    this.radius = radius;
    draw = function()
    {
        console.log('draw');
    }
}

const circle = new Circle(1)
console.log(Circle.prototype);
```

Prototype Members vs Instance Members

```
function Circle(radius)
{
    this.radius = radius;
    this.draw = function()
    {
        console.log('draw');
    }
}

const c1 = new Circle(1);
const c2 = new Circle(2);
```

if we have thousand circle objects in a memory, we will have thousand copies of draw method in memory. This leads to lot of memory wastage.

We can take out draw method out of its object and put it in its prototype.

```
function Circle(radius)
{
    //Instance Members
    this.radius = radius;
}

//Prototype Members
Circle.prototype.draw = function()
{
    console.log('draw');
}

//Method Overriding
Circle.prototype.toString = function()
{
    return `Radius of this circle is ${this.radius}`;
}

const c1 = new Circle(1);
const c2 = new Circle(2);

console.log(c1.toString()); //Radius of this circle is 1
```

Iterating Prototype Members vs Instance Members

```
// Instance Members
for(let key of Object.keys(c1))
{
    console.log(key);
}

// radius

// Instance Members + Prototype Members
for(let key in c1)
{
    console.log(key);
}

//radius
//draw
//toString
```

Avoid Extending the Built-in Objects

```
Array.prototype.shuffle = function()
{
    // Some Logic
}
```

We should not modify the built in objects in Javascript. If we are using a library and in that library someone also has extended the array prototype and added the shuffle method but with different implementation, it will become very difficult to debug the problem. Also, Javascipt developers can add this method in arrays in future.

Don't modify objects you don't own!