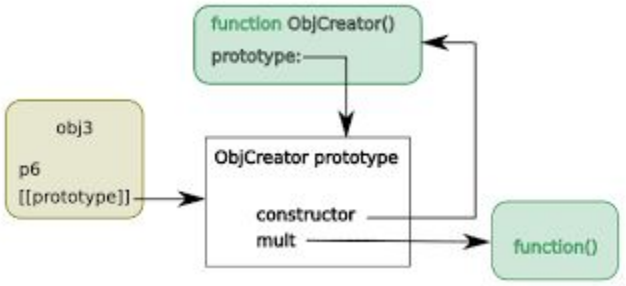
Java Script objects and Prototypes



Object Creation and Modification

- JavaScript is an object-based language. Everything is an object in JavaScript.
- JavaScript is template based not class based. Here, we don't create class to get the object. But, we direct create objects.
- In JavaScript, an object is an unordered collection of key-value pairs. Each key-value pair is called a property.
- An object can be created with figure brackets {…} with an optional list of properties. A property is a "key: value" pair, where a key is a string (also called a "property name"), and value can be anything.
- Properties of an object are accessed using a dot notation: *object.property*
- Properties are not variables, so they are not declared
- The new expression is used to create an object
  - This includes a call to a *constructor*
  - The new operator creates a blank object, the constructor creates and initializes all properties of the object

Object creation

```
var person = {
    firstName: 'Ram',
    lastName: 'Mohan'
};
console.log(person.firstName);
console.log(person.lastName);
```

```
{ firstName: 'Ram', lastName: 'Mohan' }
```

Modifying the value of a property

```
let person = {
    firstName: 'Ram',
    lastName: 'Mohan'
};

person.lastName = 'Kumar';
console.log(person);
```

```
{ firstName: 'Ram', lastName: 'Kumar' }
```

Object creation

- There are two ways to create a JavaScript object:
  - Constructor functions
  - Literal notation

```
function myObject(){
};
```

```
var myObject = {
};
```

- Defining Methods and Properties

Constructor version:

```
function myObject(){
    this.iAm = 'an object';
    this.whatAmI = function(){
        alert('I am ' + this.iAm);
    };
};
```

```
var myNewObject = new myObject();
myNewObject.whatAmI();
```

Need to instantiate (create a new instance of) the object first;

Literal version:

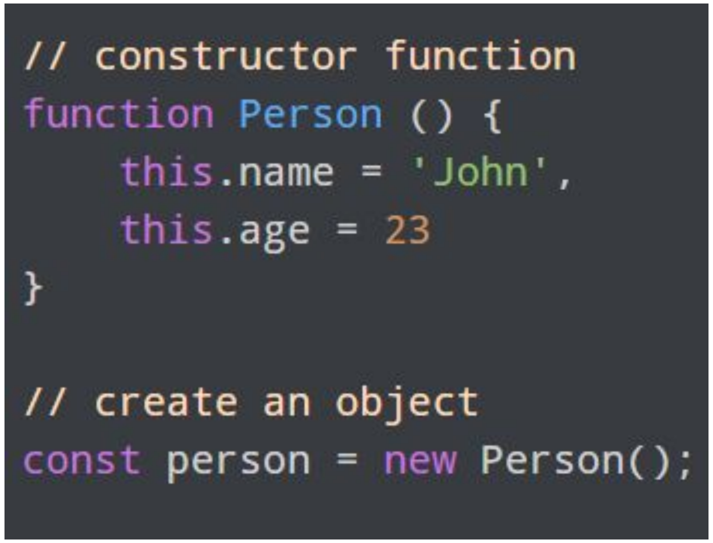
```
var myObject = {
    iAm : 'an object',
    whatAmI : function(){
        alert('I am ' + this.iAm);
    }
}
```

```
myObject.whatAmI();
```

Simply use it by referencing its variable name

Functions as constructors

- In JavaScript, any function can be used as a constructor!
  - by convention, constructors' names begin in uppercase
  - when a function is called with **`new`**, it implicitly returns `this`
  - all global "classes" (`Number`, `String`, etc.) are functions acting as constructors, that contain useful properties.



```
// constructor function
function Person () {
    this.name = 'John',
    this.age = 23
}

// create an object
const person = new Person();
```

In the above example, **`function Person()`** is an object constructor function.
To create an object from a constructor function, we use the **`new`** keyword.

Instantiating an Object: The new Operator

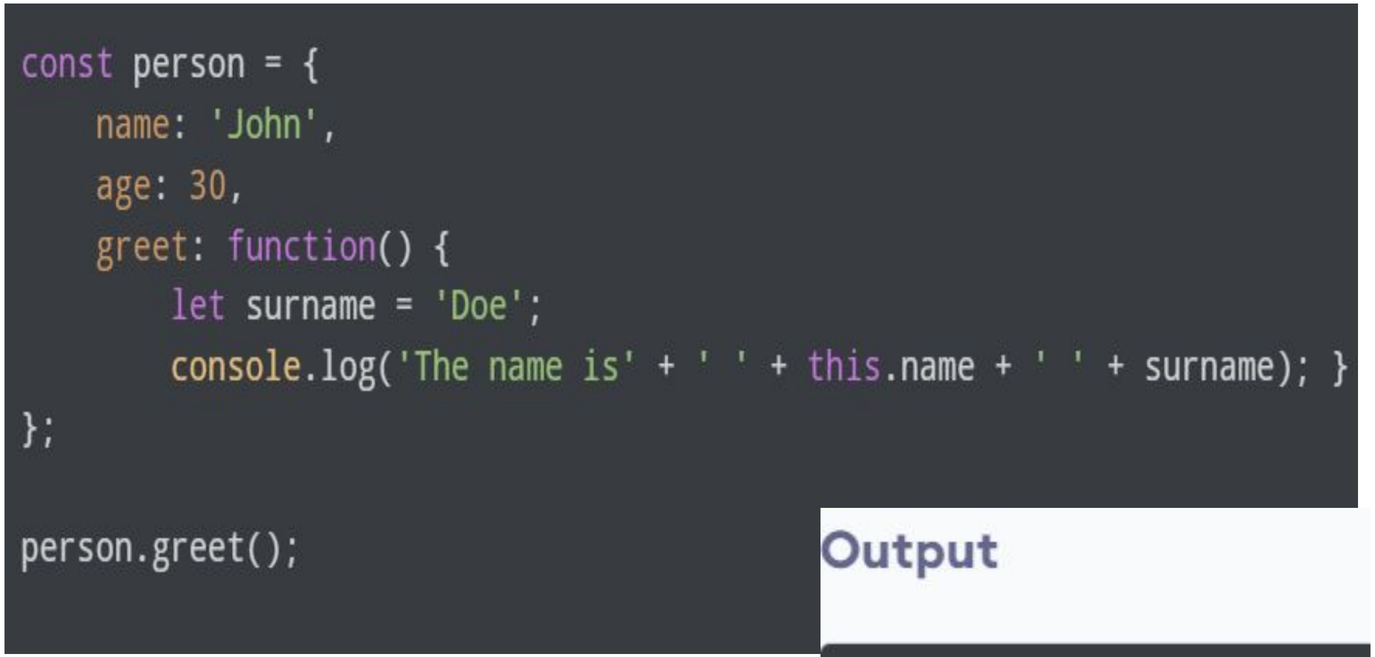
- The new keyword performs following four tasks:
  - It creates new empty object e.g. obj = { };
  - It sets new empty object's invisible 'prototype' property to be the constructor function's visible and accessible 'prototype' property. (Every function has visible 'prototype' property whereas every object includes invisible 'prototype' property)
  - It binds property or function which is declared with this keyword to the new object.
  - It returns newly created object unless the constructor function returns a non-primitive value (custom JavaScript object). If constructor function does not include return statement then compiler will insert 'return this;' implicitly at the end of the function. If the constructor function returns a primitive value then it will be ignored.

this Member

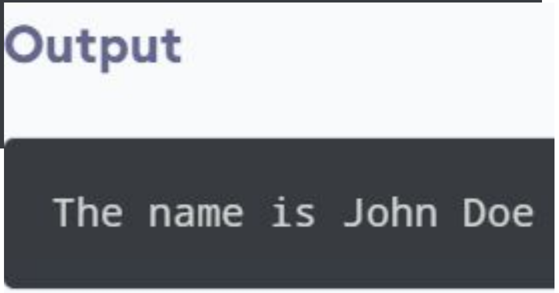
- Every time you create a new class, there is a member automatically created for it. This member is called **this** and it represents the class itself. Furthermore, the **this** member has direct access to all properties of its parent class.
- Using **this** in a constructor, you can create a list of the properties of the class. To do this, type **this**, followed by the period operator, followed by the name of the new property you want to create. To complete the creation of the property, you can assign it to an argument passed in the parentheses.
- In JavaScript, **this** keyword when used with the object's method refers to the object. this is bound to an object.

Example:

- The function inside of an object can access it's variable in a similar way as a normal function would. For example



```
const person = {
    name: 'John',
    age: 30,
    greet: function() {
        let surname = 'Doe';
        console.log('The name is' + ' ' + this.name + ' ' + surname); }
};

person.greet();
```



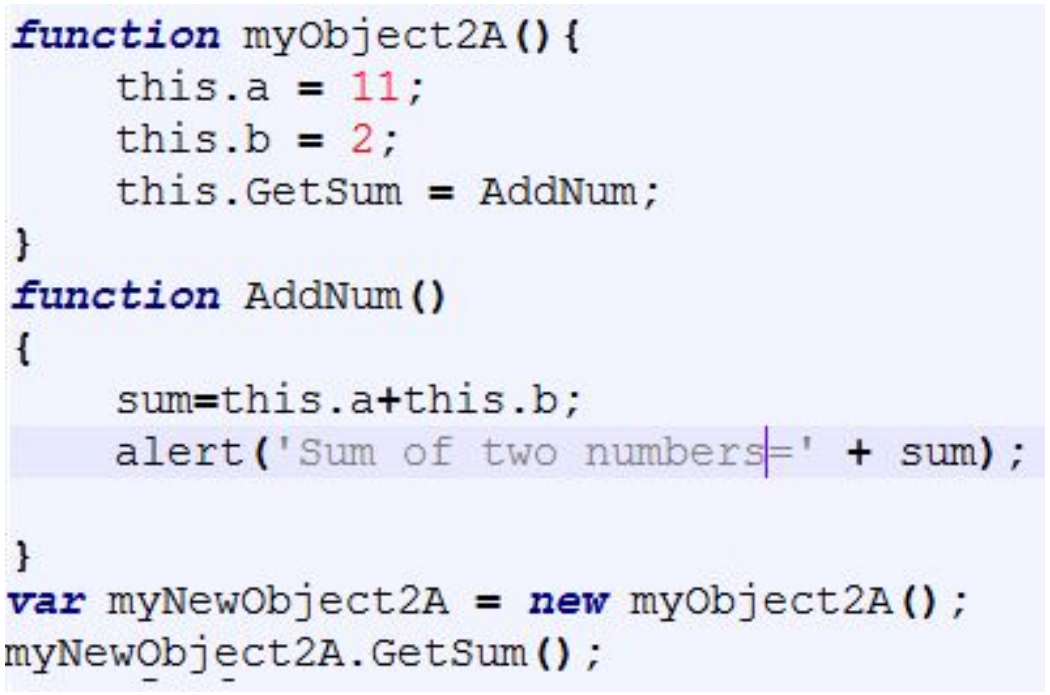
Output

```
The name is John Doe
```

Methods

- While the properties are used to describe or characterize an object, some objects may be able to perform actions. This action also is referred to as a method.
- **Creating a Method**
- A method of a class is created like a function. Since a function cannot be created inside of another function, in this case inside of the class' constructor, when defining the function, you can indicate that it belongs to the class with the use of the this object and the access to the defined member(s) of the class.
- After creating the method, to add it to your class, assign it to the desired member name using the this object in the constructor of the class as we did previously.

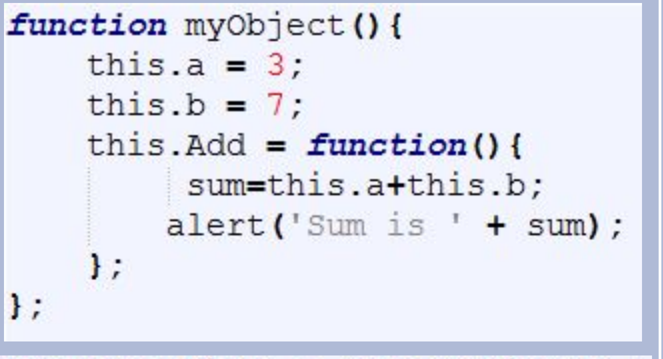
Example: adding of two numbers



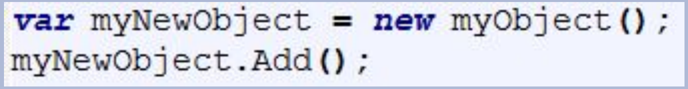
```javascript
function myObject2A(){
    this.a = 11;
    this.b = 2;
    this.GetSum = AddNum;
}
function AddNum()
{
    sum=this.a+this.b;
    alert('Sum of two numbers=' + sum);

}
var myNewObject2A = new myObject2A();
myNewObject2A.GetSum();
```

Adding of two numbers-constructor version

Passing without parameters

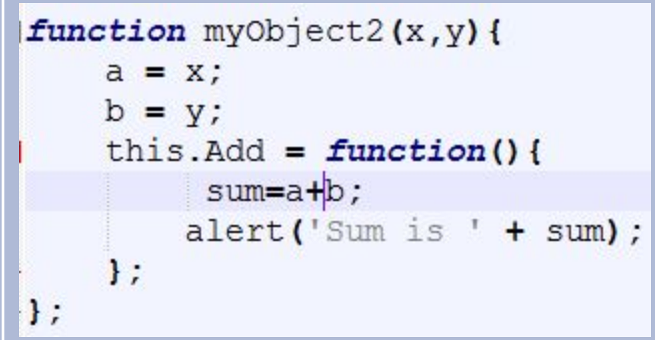


```
function myObject(){
    this.a = 3;
    this.b = 7;
    this.Add = function(){
        sum=this.a+this.b;
        alert('Sum is ' + sum);
    };
};
```

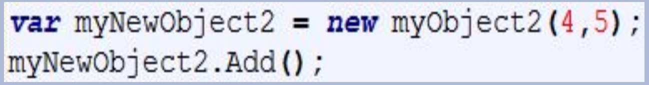


```
var myNewObject = new myObject();
myNewObject.Add();
```

Passing with parameters

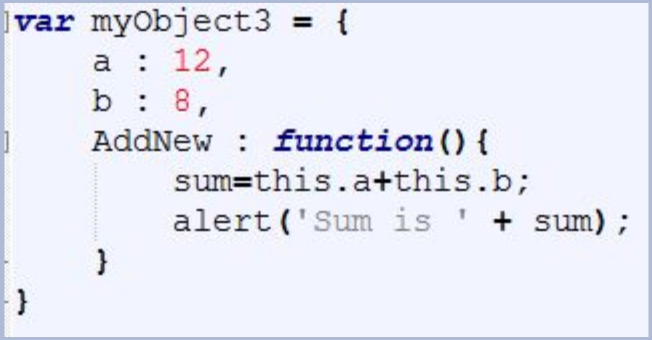


```
function myObject2(x,y){
    a = x;
    b = y;
    this.Add = function(){
        sum=a+b;
        alert('Sum is ' + sum);
    };
};
```

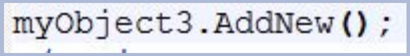


```
var myNewObject2 = new myObject2(4,5);
myNewObject2.Add();
```

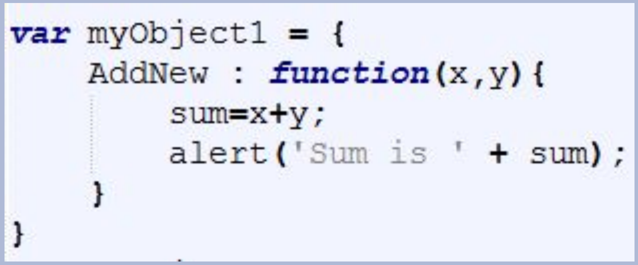
Adding of two numbers-Literal version



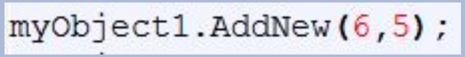
```
var myObject3 = {
    a : 12,
    b : 8,
    AddNew : function(){
        sum=this.a+this.b;
        alert('Sum is ' + sum);
    }
}
```



```
myObject3.AddNew();
```



```
var myObject1 = {
    AddNew : function(x,y){
        sum=x+y;
        alert('Sum is ' + sum);
    }
}
```



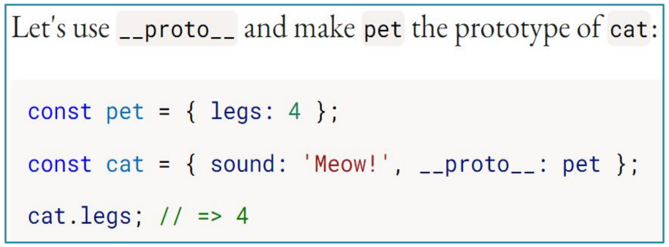
```
myObject1.AddNew(6,5);
```

Differences:

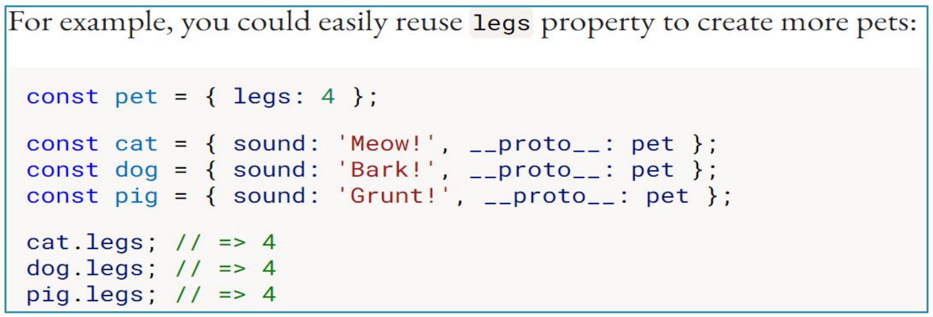
- The constructor object has its properties and methods defined with the keyword 'this' in front of it, whereas the literal version does not.
- In the constructor object the properties/methods have their 'values' defined after an equal sign '=' whereas in the literal version, they are defined after a colon ':'.
- The constructor function can have (optional) semi-colons ';' at the end of each property/method declaration whereas in the literal version if you have more than one property or method, they MUST be separated with a comma ',' and they CANNOT have semi-colons after them, otherwise JavaScript will return an error.

JavaScript Prototype

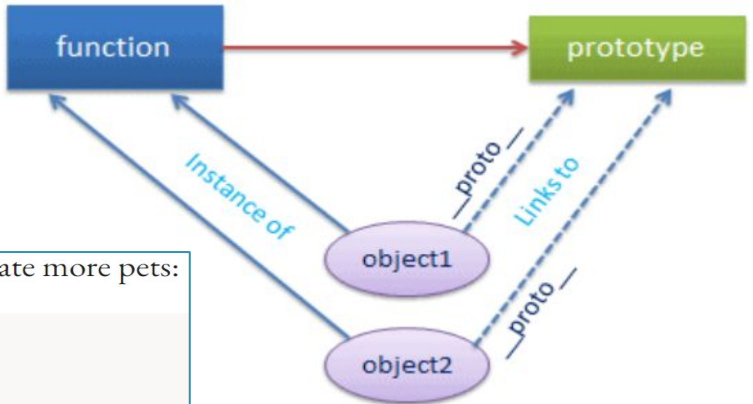
- Every JavaScript function has a prototype property (this property is empty by default), and you attach properties and methods on this prototype property when you want to implement inheritance.
- The prototype property is used primarily for inheritance; you add methods and properties on a function's prototype property to make those methods and properties available to instances of that function.
- Every object which is created using literal syntax or constructor syntax with the new keyword, includes __proto__ property that points to prototype object of a function that created this object.


Let's use `__proto__` and make `pet` the prototype of `cat`:

```
const pet = { legs: 4 };

const cat = { sound: 'Meow!', __proto__: pet };

cat.legs; // => 4
```


For example, you could easily reuse `legs` property to create more pets:

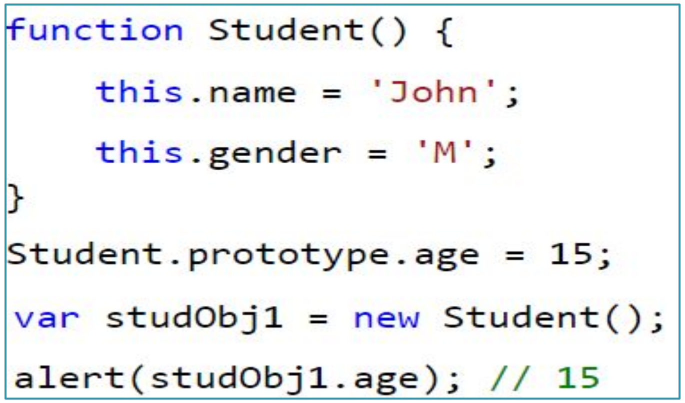
```
const pet = { legs: 4 };

const cat = { sound: 'Meow!', __proto__: pet };
const dog = { sound: 'Bark!', __proto__: pet };
const pig = { sound: 'Grunt!', __proto__: pet };

cat.legs; // => 4
dog.legs; // => 4
pig.legs; // => 4
```



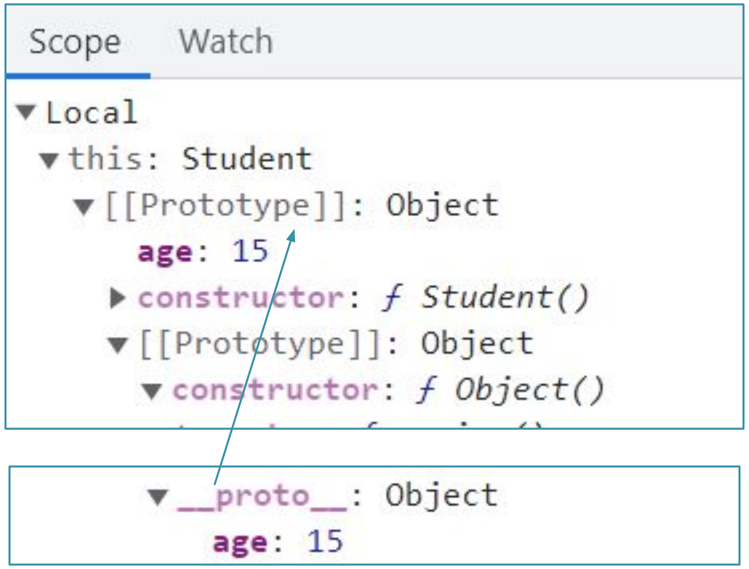
Note: `__proto__` *is deprecated, In production code* *[Object.create()]* *is recommended.*

Functions and prototypes

- The prototype is an object that is associated with every functions and objects by default in JavaScript, where function's prototype property is accessible and modifiable and object's prototype property (aka attribute) is not visible.
- The prototype object is special type of enumerable object to which additional properties can be attached to it which will be shared across all the instances of it's constructor function.
- So, use prototype property of a function in the above example in order to have age properties across all the objects as shown below.



```
function Student() {
    this.name = 'John';
    this.gender = 'M';
}
Student.prototype.age = 15;
var studObj1 = new Student();
alert(studObj1.age); // 15
```

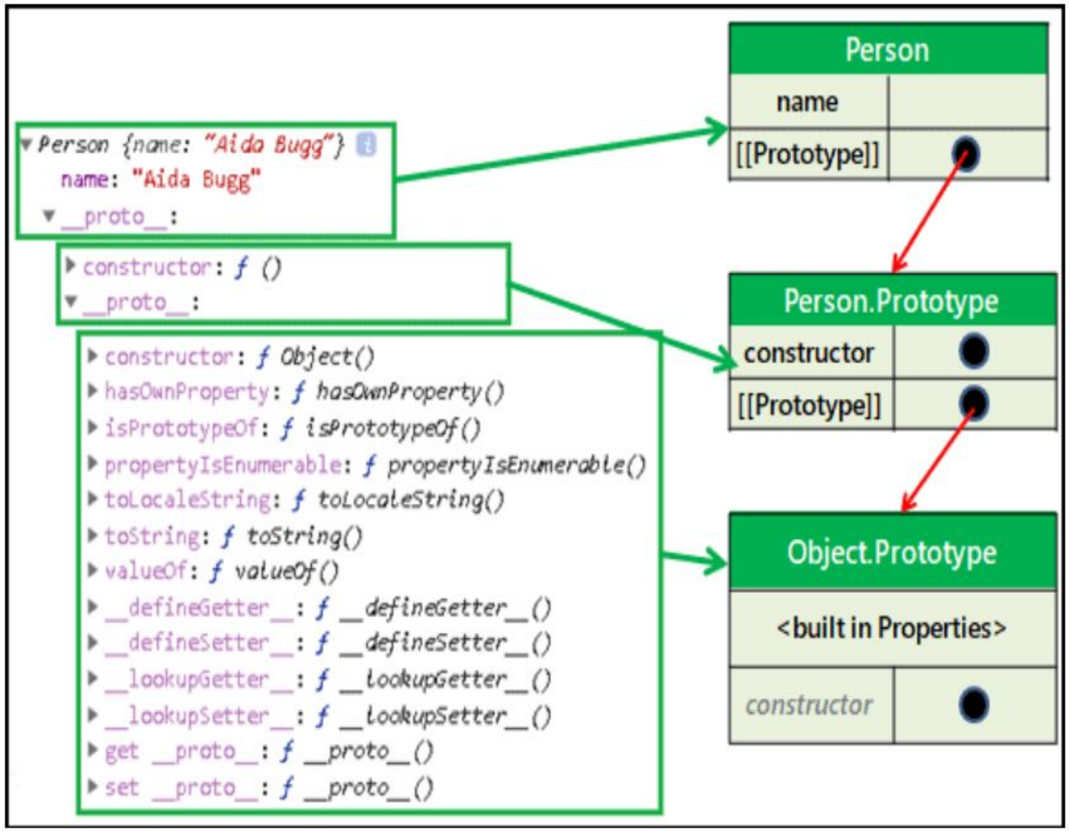


```
Scope    Watch
▼ Local
  ▼ this: Student
    ▼ [[Prototype]]: Object
        age: 15
      ▶ constructor: ƒ Student()
      ▼ [[Prototype]]: Object
        ▼ constructor: ƒ Object()

          ▼ __proto__: Object
              age: 15
```

Prototype chain

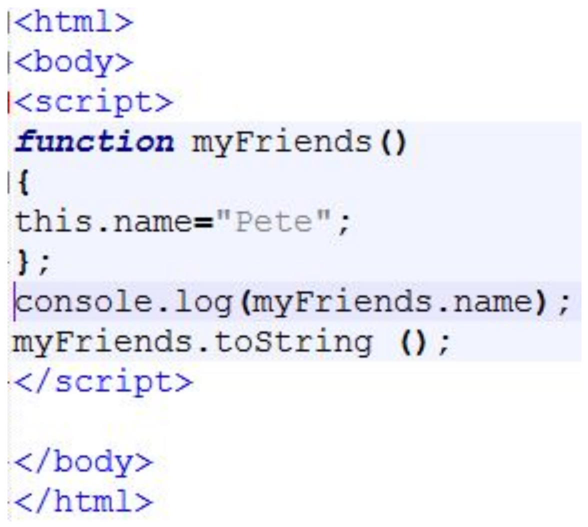
- An Object has a prototype. A prototype is also an object. Hence Even it may have its own prototype object. This is referred to as **prototype chain.**
- The following example creates a person object from the Person constructor function. In such a case the <functionName>.prototype i.e. Person.prototype becomes the Prototype of the newly created person object.

```

var Person= function() {
  this.name="Aida Bugg"
}

var person = new Person();

console.log(person);

```



```
▾ Person {name: "Aida Bugg"}
    name: "Aida Bugg"
  ▾ __proto__:
    ▸ constructor: ƒ ()
    ▾ __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__()
```

This example demonstrates the prototype chain of an object's prototype:



```
<html>
<body>
<script>
function myFriends()
{
this.name="Pete";
};
console.log(myFriends.name);
myFriends.toString ();
</script>

</body>
</html>
```

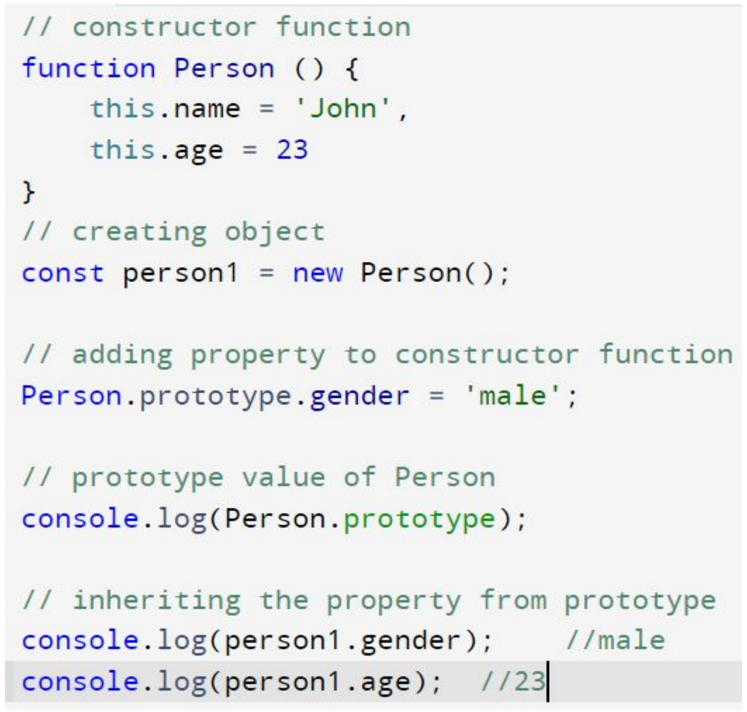
- To find the name property, the search will begin directly on the myFriends object and will immediately find the name property because we defined the property name on the myFriend object. This could be thought of as a prototype chain with one link.
- In this example, the search for the toString() method will also begin on the myFriends' object, but because we never created a toString method on the myFriends object, the compiler will then search for it on the myFriends prototype (the object which it inherited its properties from).
- And since all objects created with the object literal inherits from Object.prototype, the toString method will be found on Object.prototype

Why is Prototype Important and When it is Used?

- Prototype is important in JavaScript because JavaScript does not have classical inheritance based on Classes and therefore all inheritance in JavaScript is made possible through the prototype property. JavaScript has a prototype-based inheritance mechanism.
- Prototype is also important for accessing properties and methods of objects. The prototype attribute (or prototype object) of any object is the "parent" object where the inherited properties were originally defined.

Prototype Inheritance

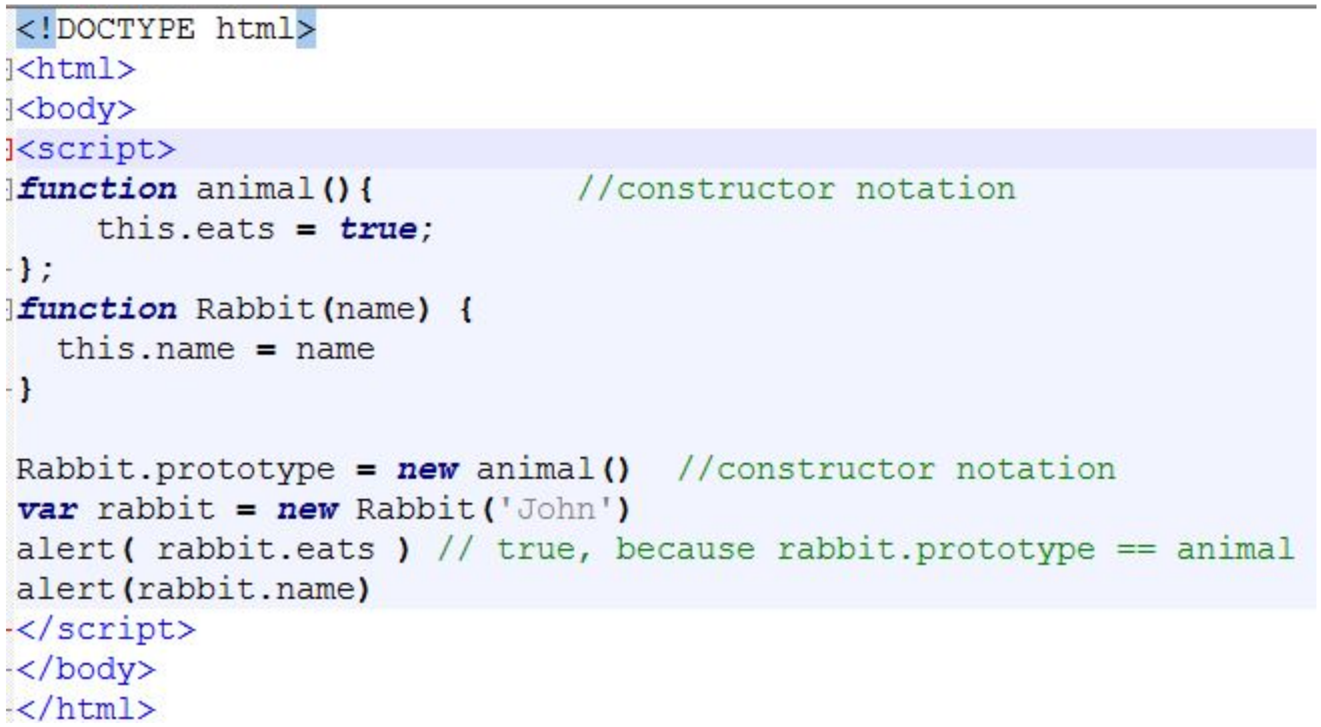
- A prototype can be used to add properties and methods to a constructor function. And objects inherit properties and methods from a prototype.



```
// constructor function
function Person () {
    this.name = 'John',
    this.age = 23
}
// creating object
const person1 = new Person();

// adding property to constructor function
Person.prototype.gender = 'male';

// prototype value of Person
console.log(Person.prototype);

// inheriting the property from prototype
console.log(person1.gender);    //male
console.log(person1.age);   //23
```

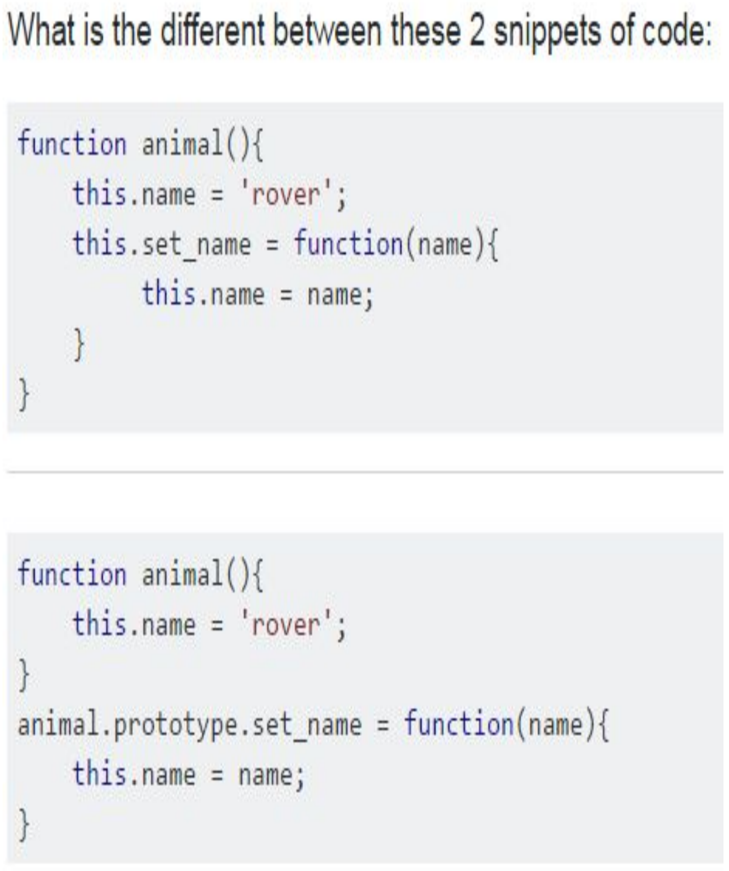
The object person1 and person2 inherits the property gender from the prototype property of Person constructor function

Example-1



```
<!DOCTYPE html>
<html>
<body>
<script>
function animal(){          //constructor notation
    this.eats = true;
};
function Rabbit(name) {
  this.name = name
}

Rabbit.prototype = new animal()  //constructor notation
var rabbit = new Rabbit('John')
alert( rabbit.eats ) // true, because rabbit.prototype == animal
alert(rabbit.name)
</script>
</body>
</html>
```

Why prototype?


What is the different between these 2 snippets of code:

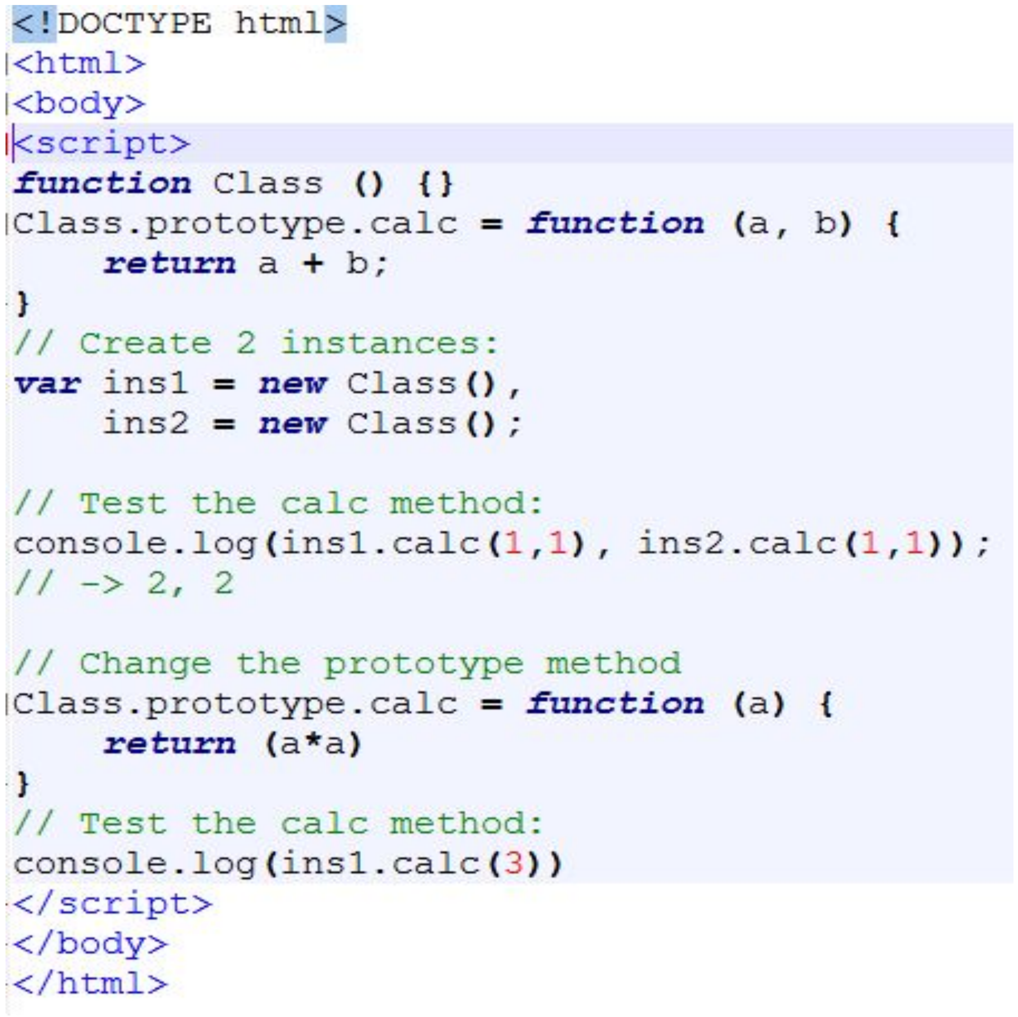
```
function animal(){
    this.name = 'rover';
    this.set_name = function(name){
        this.name = name;
    }
}
```

```
function animal(){
    this.name = 'rover';
}
animal.prototype.set_name = function(name){
    this.name = name;
}
```

- The set_name function is created each and every time you create an animal.
- In the first example, each separate animal has an own property for the set_name function, while in the second example they share the same function via their prototype.
- The advantage of the first version is that the methods can access local (private) variables declared inside the constructor.
- The advantage of the second method is that it needs less memory (since you only store the method once instead of a many times) and is more performative in current JS engines.
- Using the second method you can also modify or add methods to a class in a way that also affects instances that were already created.

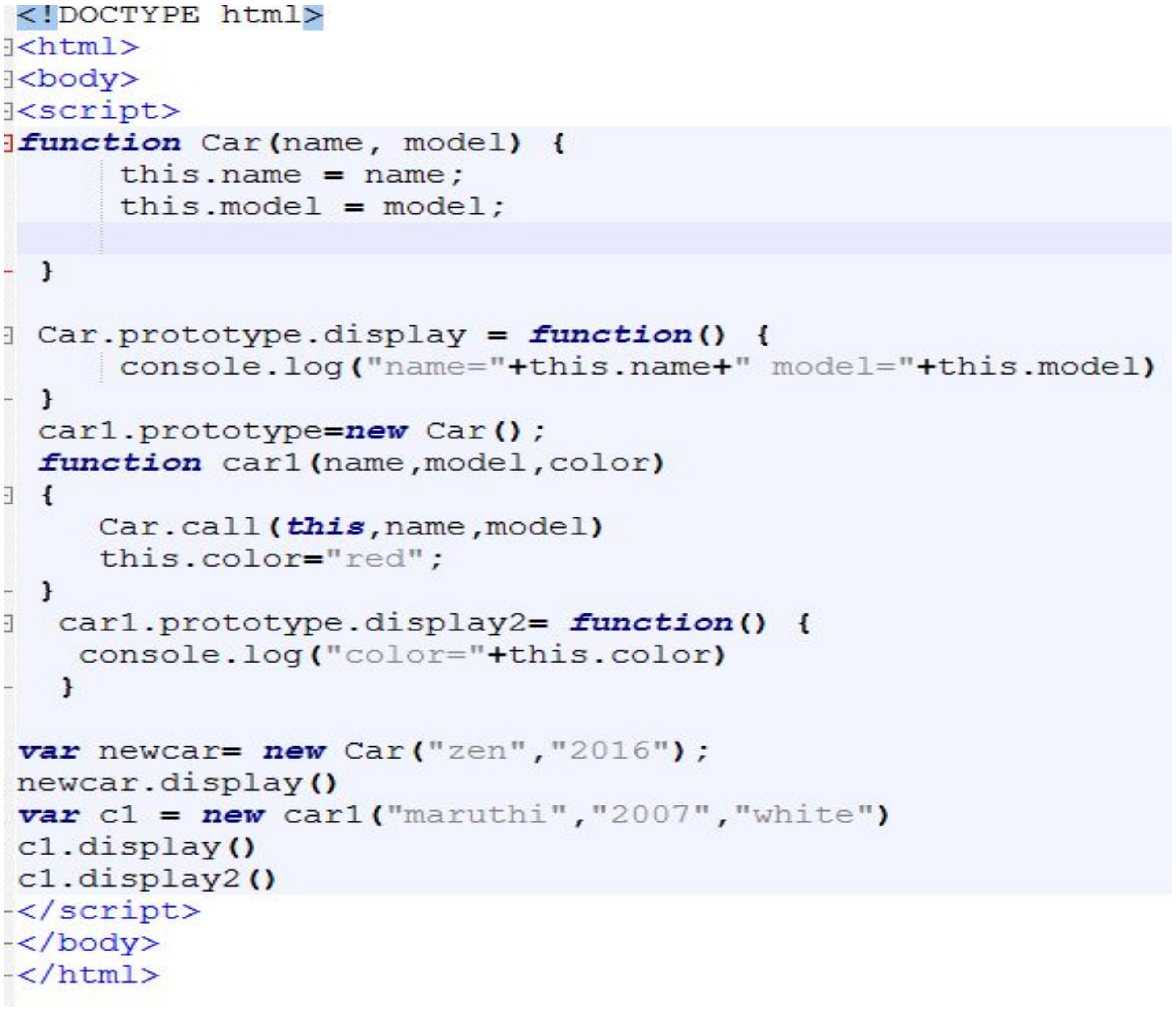
- The function does not have to be re-created each time; it exists in one place in the prototype. So when you call someAnimal.set_name("Ubu");
  The 'this' context will be set to someAnimal and set_name function will be called.
- Using the prototype makes for faster object creation, since that function does not have to be re-created each time a new object is created.

Change the prototype method



```
<!DOCTYPE html>
<html>
<body>
<script>
function Class () {}
Class.prototype.calc = function (a, b) {
    return a + b;
}
// Create 2 instances:
var ins1 = new Class(),
    ins2 = new Class();

// Test the calc method:
console.log(ins1.calc(1,1), ins2.calc(1,1));
// -> 2, 2

// Change the prototype method
Class.prototype.calc = function (a) {
    return (a*a)
}
// Test the calc method:
console.log(ins1.calc(3))
</script>
</body>
</html>
```

Multiple Levels of Inheritance



```
<!DOCTYPE html>
<html>
<body>
<script>
function Car(name, model) {
     this.name = name;
     this.model = model;

 }

 Car.prototype.display = function() {
     console.log("name="+this.name+" model="+this.model)
 }
 car1.prototype=new Car();
 function car1(name,model,color)
 {
    Car.call(this,name,model)
    this.color="red";
 }
  car1.prototype.display2= function() {
   console.log("color="+this.color)
  }

var newcar= new Car("zen","2016");
newcar.display()
var c1 = new car1("maruthi","2007","white")
c1.display()
c1.display2()
</script>
</body>
</html>
```

Conclusion

- Objects are an extremely useful and versatile feature of the JavaScript programming language. They are some of the main building blocks of writing code in JavaScript, and are a practical way to organize related data and functionality. To-do lists, shopping carts, user accounts, and locations on a webmap are all a few of the many examples of real-world JavaScript objects that you might encounter.
- JavaScript is the most common of the prototype-capable languages, and its capabilities are relatively unique. When used appropriately, prototypical inheritance in JavaScript is a powerful tool that can save hours of coding.
- Prototypes and prototypical inheritance are commonly used in many web application frameworks, such as AngularJS, to allow sharing of common behavior and state among similar components.

Exercises

- Write a JavaScript program to delete the rollno property from the following object. Also print the object before or after deleting the property.

```
var student = {
name :“Ram Mohan",
Sem : "VI",
rollno : 62 };
```

- Write a js program using prototype to calculate square,cube and square root of a number
- Write a js program using prototype to calculate area, and circumference of rectangle, square.
- Write a js program using prototype to implement a calculator.