



JAVASCRIPT OOP

banuprakashc@yahoo.co.in



Objectives

- Introduction OO concepts in JavaScript
- Define the Objects and properties
- Types of Enumerating properties of an object

Object-oriented programming

- Object-oriented programming is a programming paradigm that uses abstraction to create models based on the real world.
 - It uses several techniques from previously established paradigms, including modularity, polymorphism, and encapsulation.
 - Today, many popular programming languages (such as Java, JavaScript, C#, C++, Python, PHP, Ruby and Objective-C) support object-oriented programming (OOP).

Object-oriented programming

- **Terminology**

- **Namespace**

- A container which allows developers to bundle all functionality under a unique, application-specific name.

- **Class**

- Defines the characteristics of the object. It is a template definition of variables and methods of an object.

- **Object**

- An Instance of a class.

- **Property**

- An object characteristic, such as color.

- **Method**

- An object capability/action, such as walk. It is a subroutine or function associated with a class.

Object-oriented programming

- **Terminology Continued..**

- **Constructor**

- A method called at the moment of instantiation of an object. It has the same name as that of the class containing it.

- **Inheritance**

- A class can inherit characteristics from another class.

- **Encapsulation**

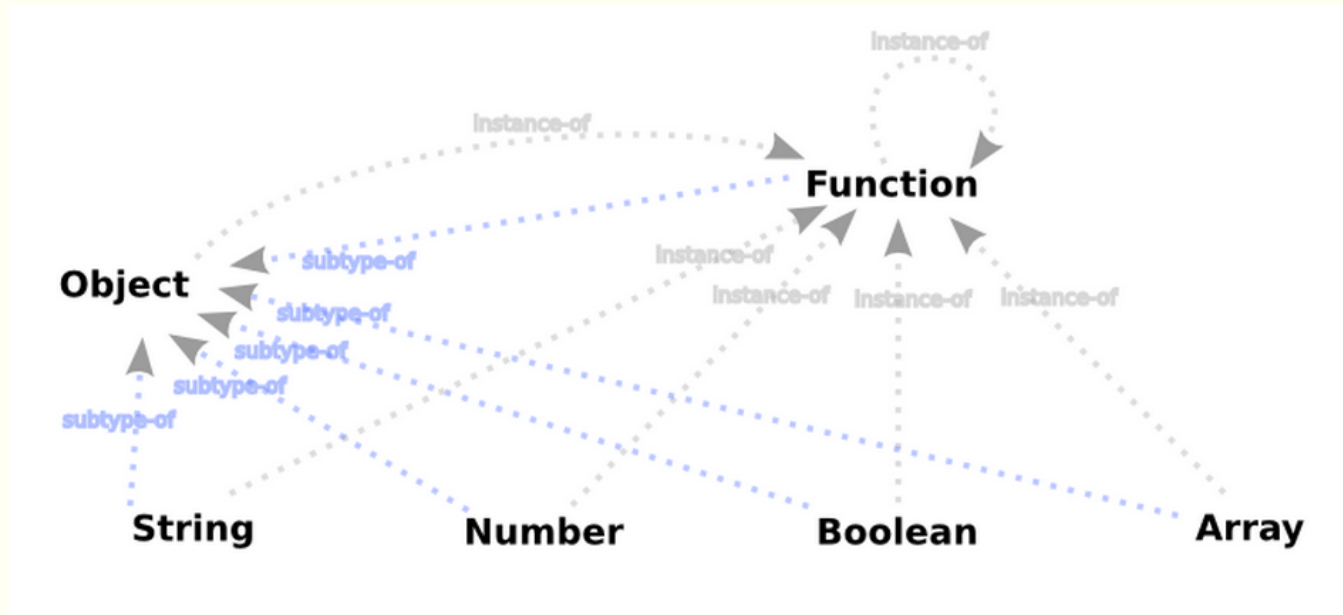
- A method of bundling the data and methods that use them together.

- **Abstraction**

- The conjunction of complex inheritance, methods, properties of an object must be able to simulate a reality model.

Custom objects

- The class
 - JavaScript is a prototype-based language which contains no class statement, such as is found in C++ or Java.
 - Instead, JavaScript uses functions as classes. Defining a class is as easy as defining a function.



Custom objects

- Constructors are used to create specific type of objects. In JavaScript, constructor functions are generally considered a reasonable way to implement instances.
- By simply prefixing a call to a constructor function with the keyword '**new**', you can tell JavaScript you would like function to behave like a constructor and instantiate a new object.

Custom Objects

- Different ways of creating objects

```
//In JavaScript, the three common ways to create new objects
//Each of the following options will create a new empty object
var firstObj = {};
var secondObj = Object.create(null);
var thirdObj = new Object();
//There are then four ways in which keys and values can then be assigned to an object:
// ECMAScript 3 compatible approaches
// 1. Dot syntax
firstObj.name = "Banu Prakash"; // write property
var authorName = firstObj.name; // access property
// 2. Square bracket syntax
secondObj['name'] = "Rahul Prakash"; // write property
var empName = secondObj['name']; // access property

// ECMAScript 5 only compatible approaches
Object.defineProperty(thirdObj, {"name":
    { value:"Smith", writable:true,enumerable:true, configurable:true},
    "age":
    {value:34, writable:true,enumerable:true, configurable:true}});
```


The constructor

- The constructor is called at the moment of instantiation (the moment when the object instance is created).
- In JavaScript the function serves as the constructor of the object, therefore there is no need to explicitly define a constructor method.
- Every action declared in the class gets executed at the time of instantiation.
- The constructor is used to set the object's properties or to call methods to prepare the object for use.
- Adding class methods and their definitions occurs using a different syntax described later.

The constructor

- Example

```
function Employee(name, age) {  
  
}  
  
var raj = new Employee("Raj", 24);  
var shyam = new Employee("Shyam", 31);
```

- Using function expression

```
var Person = function() {  
    console.log("instance created");  
}  
  
var firstPerson = new Person();  
  
var secondPerson = new Person();
```

Object attributes

- The Property
 - Properties are variables contained in the class; every instance of the object has those properties.
 - Properties are set in the constructor (function) of the class so that they are created on each instance.
- The “this” keyword:
 - Refers to the current object.
 - Working with properties from within the class is done using the keyword “this”.

Enumerating properties of an object

- Print all the properties of an Object

```
var Book = function(id, title, price) {  
    this.id = id;  
    this.title = title;  
    this.price = price;  
}  
  
var jsBook = new Book(100, "Head First JavaScript", 543.50);  
  
for (property in jsBook) {  
    console.log(property);  
}
```



id
title
price

Accessing property of an object

- Accessing (reading or writing) a property outside of the class is done with the Syntax: **instanceName.property**;
- this is the same syntax used by C++, Java, and a number of other languages

```
var Book = function(id, title, price) {  
    this.id = id;  
    this.title = title;  
    this.price = price;  
}  
  
var jsBook = new Book(100, "Head First JavaScript", 543.50);  
  
console.log(jsBook.title); // Head First JavaScript  
console.log(jsBook.price); // 543.50
```

The methods

- Methods follow the same logic as properties; the difference is that they are functions and they are defined as functions.
- Calling a method is similar to accessing a property, but you add () at the end of the method name, possibly with arguments.
- To define a method, assign a function expression using "this.functionName" to a instance method.

Example using Constructor Pattern

```
//constructor pattern
function Employee(name, age) {
    this.name = name; //the keyword 'this' references the new object that's being created
    this.age = age;
    this.getName = function() {
        return name;
    }
    this.getAge = function() {
        return age;
    }
}
var firstEmp = new Employee("Smith" , 45);
var secEmp = new Employee("Diana", 22);
console.log(firstEmp.getName() + ", " + firstEmp.getAge());
```

Deleting Properties

- The only way to remove a property from an **object** is to use the delete operator.
- Setting the property to undefined or null only removes the value associated with the property, but not the key.

```
var Book = function(id, title, price) {  
    this.id = id;  
    this.title = title;  
    this.price = price;  
}  
  
var jsBook = new Book(100, "Head First JavaScript", 543.50);  
  
for (property in jsBook) {  
    console.log(property); // id, title, price  
}  
  
delete jsBook.price;  
  
for (property in jsBook) {  
    console.log(property); // id, title  
}
```


Comparing Objects

- While both `==` and `===` are called equality operators, they behave differently when at least one of their operands is an Object.
- Here, both operators compare for identity and not equality; that is, they will compare for the same instance of the object.

Comparing Objects

- Equality checking by providing our own custom method [equals]

```
var Book = function(id, title, price) {  
  this.id = id;  
  this.title = title;  
  this.price = price;  
  
  this.equals = function(other)  
  {  
    if ( (id === other.id)  
      && (title === other.title)  
      && (price === other.price)) {  
      return true;  
    } else {  
      return false;  
    }  
  }  
}  
  
var jsBook = new Book(100, "Head First JavaScript", 543.50);  
  
var scriptBook = new Book(100, "Head First JavaScript", 543.50);  
  
jsBook.equals(scriptBook); //true
```

The instanceof Operator

- The instanceof operator compares the constructors of its two operands.
- It is useful when comparing custom made objects.

```
var Book = function(id, title, price) {
  this.id = id;
  this.title = title;
  this.price = price;

  this.equals = function(other)
  {
    if ( (id === other.id)
        && (title === other.title)
        && (price === other.price)) {
      return true;
    } else {
      return false;
    }
  }
}

var Dummy = function(id,title,price) {
}

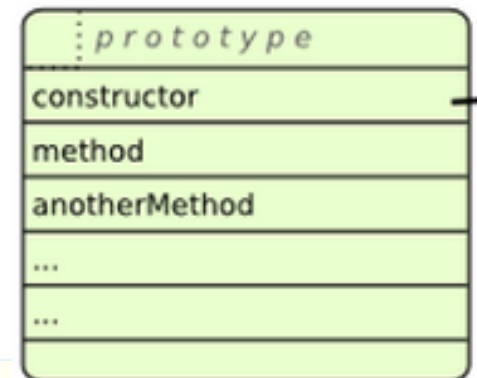
var jsBook = new Book(100, "Head First JavaScript", 543.50);

jsBook instanceof Book //true
jsBook instanceof Dummy // false
```

The function prototype

- The `Function.prototype` property represents the Function prototype object.
- Function objects inherit from `Function.prototype`.
- **`Function.prototype.constructor`**
- Specifies the function that creates an object's prototype.
- All objects inherit a `constructor` property from their prototype.

```
var obj = {};  
console.log("obj instance of object :" + (obj.constructor === Object));  
  
var values = [];  
console.log("values is an array :" + (values.constructor === Array));  
  
var ten = new Number(10);  
console.log("values is an array :" + (ten.constructor === Number));
```



Function Prototype Methods

- `Function.prototype.apply()`
 - The `apply()` method calls a function with a given **this** value and **arguments** provided as a array.
 - Syntax: `fun.apply(this, [argsArray])`
 - Use `apply()` to chain constructors
- `Function.prototype.call()`
 - The `call()` method calls a function with a given **this** value and **arguments** provided as a individually.
 - Syntax: `fun.call(this[, arg1[, arg2[, ..]]])`

Function Prototype Methods

- Both `call()` and `apply()` perform similar functions:
 - They execute functions in the context, or scope of the first argument that you pass to them.
 - They can only be called on other functions.

```
var Book = function(id, title, price) {  
    this.id = id;  
    this.title = title;  
    this.price = price;  
}  
  
var info = function() {  
    return this.id + " : " + this.title + " : " + this.price;  
}  
  
var jsBook = new Book(100, "Head First JavaScript", 543.50);  
var javaBook = new Book(101, "Complete Ref", 650.00);  
  
info.call(jsBook);  
info.call(javaBook);
```

Function Prototype Methods

- The call has limitations when you don't know the number of arguments as shown in dispatch.

```
var person1 = {  
  name : 'Rahul',  
  age : 16,  
};  
var person2 = {  
  name : 'Kavitha',  
  age : 42,  
};  
  
var say = function(greeting) {  
  alert(greeting + this.name);  
};  
  
var update = function(name, age){  
  this.name = name;  
  this.age = age;  
};
```

```
var dispatch = function(person, method, args){  
  method.apply(person, args);  
};  
  
dispatch(person1, say, ['Hello, ']);  
dispatch(person2, update, ['Kavitha Banu', 42]);  
  
say.call(person1, "Hi, ");  
say.call(person2, "Hellooooooooo., ");
```

Constructor pattern with Prototypes

- Functions in JavaScript have a property called a prototype.
- When you call a JavaScript constructor to create an object, all the properties of the constructor's prototype are then made available to the new object.

```
//constructor pattern with prototype
function Employee(name, age) {
    this.name = name; //the keyword 'this' references the new object that's being created
    this.age = age;
}
Employee.prototype.getName = function() {
    return this.name;
}
Employee.prototype.getAge = function() {
    return this.age;
}

var firstEmp = new Employee("Smith" , 45);
var secEmp = new Employee("Diana", 22);

console.log(firstEmp.getName() + ", " + firstEmp.getAge());
```


Constructor pattern with Prototypes

- Another approach of using prototype

```
var Book = function(id, title, price) {  
    this.id = id;  
    this.title = title;  
    this.price = price;  
}  
  
Book.prototype = {  
    getTitle : function() {  
        return this.title;  
    },  
    getPrice : function() {  
        return this.price;  
    }  
}  
  
var jsBook = new Book(100, "Head First JavaScript", 543.50);  
jsBook.getTitle(); // Head First JavaScript
```

Instance owned Vs. class-owned methods

```
function Blog(body, date) {  
  /*  
    The this keyword is used to set properties  
    and methods that are owned by an instance  
  */  
  this.body = body;  
  this.date = date;  
  /*  
    Every instance of Blog gets  
    its own copy of these methods  
  */  
  this.getBlog = function() {  
    return this.body + "," + this.date;  
  };  
}
```

```
function Blog(body, date) {  
  /*  
    The this keyword is used to set properties  
    and methods that are owned by an instance  
  */  
  this.body = body;  
  this.date = date;  
}  
/*  
  Own once, run many: class-owned methods.  
  Storing a method in a class allows all instances to share one copy.  
  Use prototype to work at a class level.  
  Since the methods aren't being assigned to a particular blog instance,  
  the assignment takes place outside of the constructor.  
*/  
Blog.prototype.getBlog = function() {  
  return this.body + "," + this.date;  
};
```

Class variable and class methods

```
/*
    Prototype object allows you to add properties and methods at class level.
    class property [ One copy for all objects]
    Prototype object is where class properties are stored.
    Class properties are created outside of an objects constructor
    with a little help from the hidden prototype object
*/
Blog.prototype.signature = "Banu Prakash";

/*
    Class methods owned by a class,
    and can only access class properties
*/
Blog.getSignature = function() {
    return Blog.prototype.signature;
}

Blog.blogSorter = function compare(blog1, blog2) {
    return blog1.date - blog2.date;
};
```

Inheritance

- Generalization and Specialization relationship
- Inheritance is not as straightforward in JavaScript as in other object-oriented languages.
- JavaScript uses object-based (prototypal) inheritance; this can be used to emulate class-based (classical) inheritance.
- Each style also has different performance characteristics, which can be an important factor in deciding which to use.

Inheritance

- Java uses class based system and JavaScript uses prototype based inheritance

JavaScript

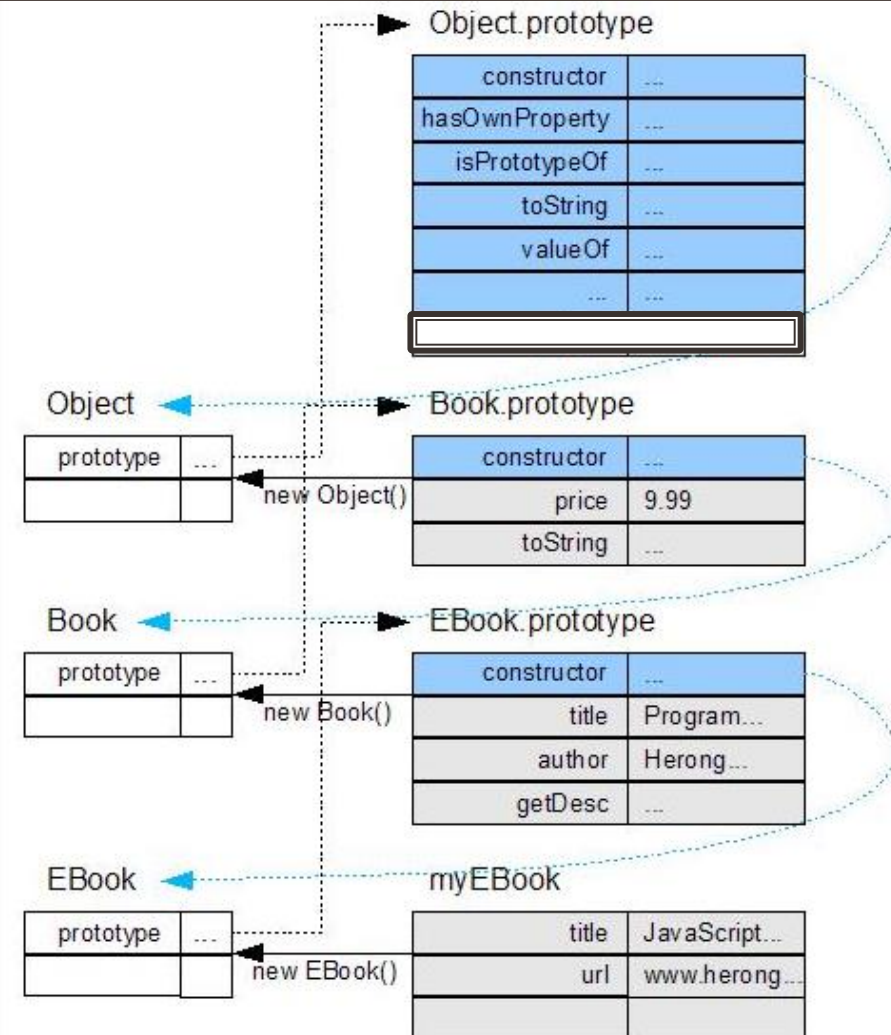
```
1 function Manager () {  
2   this.reports = [];  
3 }  
4 Manager.prototype = new Employee;  
5  
6 function WorkerBee () {  
7   this.projects = [];  
8 }  
9 WorkerBee.prototype = new Employee;
```

Java

```
1 public class Manager extends Employee {  
2   public Employee[] reports;  
3   public Manager () {  
4     this.reports = new Employee[0];  
5   }  
6 }  
7  
8 public class WorkerBee extends Employee {  
9   public String[] projects;  
10  public WorkerBee () {  
11    this.projects = new String[0];  
12  }  
13 }
```

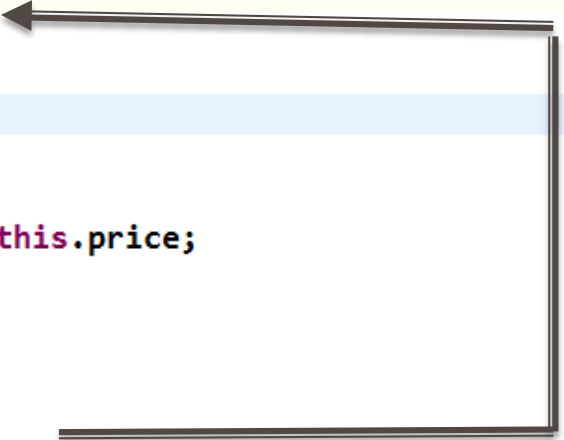
Prototypal inheritance

- Objects inherit from objects
- An Object contains a link to another object
- Delegation. Differential inheritance
`var newObject = Object.create(oldObject);`



Inheritance Example

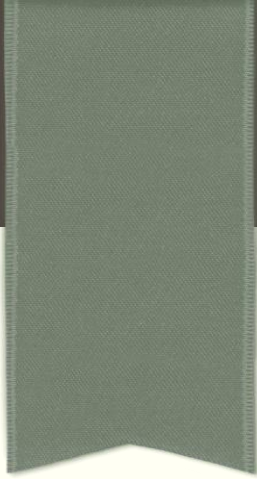
```
function Product(name, price) {  
    this.name = name;  
    this.price = price;  
}  
Product.prototype.toString = function() {  
    return "Product: " + this.name + ", " + this.price;  
}  
  
function Mobile(name, price, connectivity) {  
    //Product.apply(this, arguments);  
    Product.call(this, name, price);  
    this.connectivity = connectivity;  
}  
// Here's where the inheritance occurs  
Mobile.prototype = new Product();  
// Otherwise instances of Mobileat would have a constructor of Product  
Mobile.prototype.constructor = Mobile;  
// override  
Mobile.prototype.toString = function() {  
    return "Mobile: " + this.name + ", " + this.price + ", " + this.connectivity;  
}  
  
var nokia = new Mobile("Nokia Lumia" , 45000.66, "3G");  
alert(nokia.toString());
```

A diagram consisting of a rectangular box with a double border. The box encloses the Mobile constructor function and its prototype chain setup. An arrow points from the top of the box to the Product constructor function, indicating the inheritance relationship where Mobile inherits from Product.

Inheritance

- Method override and invoking super type methods

```
// override
Mobile.prototype.toString = function() {
    var txt = Product.prototype.toString.call(this);
    return "Product details : " + txt + ", Mobile Details: " + this.connectivity;
}
```

REFERENCES

Contains the reference that will supplement the self learning and will be needed for completing the assignments & practice questions

Reference

- <http://www.javascriptkit.com/javatutors/oopjs.shtml>