CONTENTS INCLUDE:

Object-Oriented JavaScript:

Advanced Techniques for Serious Web Applications

By Ibrahim Levent

As HTML5 standards mature, JavaScript has become the de-facto programming language of the web for client-side environments, and is gaining ground rapidly on the server-side. As JavaScript code-bases grow larger and more complex, the object-oriented capabilities of the language become increasingly crucial. Although JavaScript's object-oriented capabilities are not as powerful as those of static server-side languages like Java, its current object-oriented model can be utilized to write better code.

In this Refcard, you'll learn how to write object-oriented code in JavaScript. Code samples include some EcmaScript 5 (JavaScript's specification) features and have been tested in Firefox 17, Chrome 23 and Internet Explorer 9.

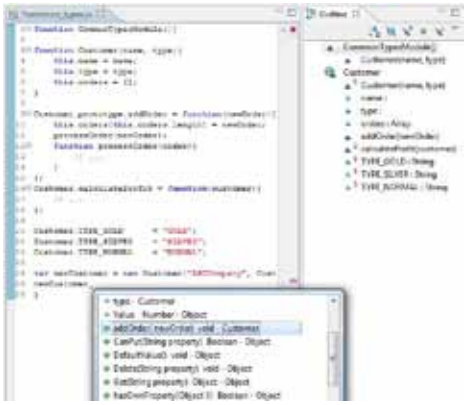
COMPARISON OF JAVA AND JAVASCRIPT

Java	JavaScript
Statically typed	Dynamically typed
Custom types are defined with class, interface or enum	Custom types are defined with functions and prototypes
Types cannot be changed at runtime	Types can be changed at runtime
Variables needs a type when declared (Strong typing)	No type needed when variables are declared (Loose typing)
Constructors are special methods	Constructor is just a function; no distinction between function and constructor
Class and instance are different entities	Everything is object; constructor functions and prototypes are also object
Static and instance members supported	No direct static member support
Abstract types supported with abstract classes and interfaces	No direct abstract type support
Good enough member scoping options; private, package, protected, public	Only public members supported by language
Rich inheritance mechanism	Poor prototypal inheritance
Method overloading and overriding possible	No direct overloading and overriding support
Rich reflection features	Some reflection features
Modularity with package support	No direct package or module support

OBJECT TYPE DEFINITION

```
function MyType(){
   if (!(this instanceof MyType))
         throw new Error("Constructor can't be called as a function");
}
var myInstance = new MyType();
MyType(); // Error: Constructor can't be called as a function
```

In Eclipse, object types can be easily inspected in the JavaScript perspective. Constructor, instance members, static members and inner functions are all separated in Outline View.



Instance Members

Instance members are accessed from instance objects which are created with the "new" operator. Instance members are created via "this" keyword, prototype, constructor closure or Object.defineProperty

```
function Cat(name){
          var voice = "Meow";
          this.name = name;
          this.say = function(){
                    return voice;
}
}
Cat.prototype.eat = function(){
          return "Eating";
}
var cat = new Cat("Fluffy");
Object.defineProperty(cat,"numLegs",{value:4,writable:true,enumerable:true,configurable:tr
ue});

console.log(cat.name); // Fluffy
console.log(cat.numLegs); // 4
console.log(cat.say()); // Meow
console.log(cat.eat()); // Eating
```

Static Members

There is no direct support for static members. Constructor functions are used to create static members. Static members can't be accessed with "this" keyword.

Public Static Members:

```
function Factory(){
}

// public static method
Factory.getType = function (){
return “Object Factory”;
};
// public static field
Factory.versionId = “F2.0”;

Factory.prototype.test = function(){
console.log(this.versionId); // undefined
console.log(Factory.versionId); // F2.0
console.log(Factory.getType()); // Object Factory
}

var factory = new Factory();
factory.test();
```

Private Static Members:

```
var Book = (function () {
          // private static field
          var numOfBooks = 0;

          // private static method
          function checkIsbn(isbn) {
                    if (isbn.length != 10 && isbn.length != 13)
                              throw new Error(“isbn is not valid!”);
          }

          function Book(isbn, title) {
                    checkIsbn(isbn);
                    this.isbn = isbn;
                    this.title = title;
                    numOfBooks++;
                    this.getNumOfBooks = function () {
                              return numOfBooks;
                    }

          }
          return Book;
})();

var firstBook = new Book(“0-943396-04-2”, “First Title”);
console.log(firstBook.title); // First Title
console.log(firstBook.getNumOfBooks()); // 1
var secondBook = new Book(“0-85131-041-9”, “Second Title”);
console.log(firstBook.title); // First Title
console.log(secondBook.title); // Second Title
console.log(firstBook.getNumOfBooks()); // 2
console.log(secondBook.getNumOfBooks()); // 2
```

Abstract Types

JavaScript is a loosely typed language, so you don't have to specify the variable type of a variable when you declare it and pass as an argument. This reduces the need for abstract types like interfaces. But abstract types may be needed to collect common functionality when inheritance is used.

```
(function(){

var abstractCreateLock = false;
// abstract type
function BaseForm(){
          if(abstractCreateLock)
                    throw new Error(“Can’t instantiate BaseForm!”);
}
BaseForm.prototype = {};
BaseForm.prototype.post = function(){
          throw new Error(“Not implemented!”);
}

function GridForm(){
}

GridForm.prototype = new BaseForm();
abstractCreateLock = true;
GridForm.prototype.post = function(){
          // ...
          return “Grid is posted.”;
}

window.BaseForm = BaseForm;
window.GridForm = GridForm;

})();

var myGrid = new GridForm();
console.log(myGrid.post()); // Grid is posted.
var myForm = new BaseForm(); // Error: Can’t instantiate BaseForm!
```

Interfaces

There is no direct interface or virtual class support in JavaScript. It can be mimicked as below, leaving a method signature check:

```
var Interface = function (name, methods) {
          this.name = name;
          // copies array
          this.methods = methods.slice(0);
};

Interface.checkImplements = function (obj, interfaceObj) {
          for (var i = 0; i < interfaceObj.methods.length; i++) {
                    var method = interfaceObj.methods[i];
                    if (!obj[method] || typeof obj[method] !==
“function”)
                              thrownewError(“Interfacenotimplemented!
Interface: “ + interfaceObj.name + “ Method: “ + method);
          }
};
var iMaterial = new Interface(“IMaterial”, [“getName”, “getPrice”]);

function Product(name,price,type){
          Interface.checkImplements(this, iMaterial);
          this.name = name;
          this.price = price;
          this.type = type;
}

Product.prototype.getName = function(){
          return this.name;
};

Product.prototype.getPrice = function(){
          return this.price;
};

var firstCar = new Product(“Super Car X11”,20000,”Car”);
console.log(firstCar.getName()); // Super Car X11
delete Product.prototype.getPrice;
varsecondCar=newProduct(“SuperCarX12”,30000,”Car”);//Error:Interfacenotimplemented!
```

Singleton Objects

If you need to use one instance of an object, use a Singleton, like this:

```
var Logger = {
          enabled:true,
          log: function(logText){
                    if(!this.enabled)
                              return;
                    if(console && console.log)
                              console.log(logText);
                    else
                              alert(logText);
          }
}

Or

function Logger(){
}
Logger.enabled = true;
Logger.log = function(logText){
          if(!Logger.enabled)
                    return;
          if(console && console.log)
                    console.log(logText);
          else
                    alert(logText);
};

Logger.log(“test”); // test
Logger.enabled = false;
Logger.log(“test”); //
```

OBJECT CREATION

With "new" Operator

Instances of built-in or user-defined object types can be created with the “new” operator. The “new” operator runs only with constructor functions. An already created object can't be used with “new” again. First, the “new” operator creates an empty object; then it calls the constructor function while assigning the newly created object as “this”. The constructor function runs with the given arguments, performing its coded initializations.

```
//or var dog = {};
//or var dog = new MyDogType();
var dog = new Object();
dog.name = "Scooby";
dog.owner = {};
dog.owner.name = "Mike";
dog.bark = function(){
          return "Woof";
};

console.log(dog.name); // Scooby
console.log(dog.owner.name); // Mike
console.log(dog.bark()); // Woof
```

With Object Literal

It is very easy to create objects with object literal. It is also possible to create nested objects.

```
var dog = {
          name:"Scoobyî",
          owner:{
                    name:"Mike"
          },
          bark:function(){
                    return "Woof";
          }
};

console.log(dog.name); // Scooby
console.log(dog.owner.name); // Mike
console.log(dog.bark()); // Woof
```

MEMBER SCOPING

Private Fields

In JavaScript, there is no built-in private field support. But this support can be achieved with constructor closures. Variables defined within a constructor function can be used as private fields. Any methods using private fields should be declared in the constructor function (note that prototype methods can't be declared in the constructor function and used with private fields). Private setter and getter functions should be declared in the constructor function.

```
function Customer(){
          // private field
          var risk = 0;

          this.getRisk = function(){
                    return risk;
          };
          this.setRisk = function(newRisk){
                    risk = newRisk;
          };
          this.checkRisk = function(){
                    if(risk > 1000)
                              return "Risk Warning";
                    return "No Risk";
          };
}

Customer.prototype.addOrder = function(orderAmount){
          this.setRisk(orderAmount + this.getRisk());
          return this.getRisk();
};

var customer = new Customer();

console.log(customer.getRisk()); // 0
console.log(customer.addOrder(2000)); // 2000
console.log(customer.checkRisk()); // Risk Warning
```

Private Methods

Also called "Nested" or "Inner Functions". Private methods are defined within another function and can't be accessed from outside. Private methods can be declared in any part of a function.

```
function Customer(name){
          var that = this;
          var risk = 0;
          this.name = name;
          this.type = findType();
          // private method
          function findType() {
                    console.log(that.name);
                    console.log(risk);
                    return "GOLD";
          }
}
```

Or

```
function Customer(name){
          var that = this;
          var risk = 0;
          this.name = name;
          // private method
          var findType = function() {
                    console.log(that.name);
                    console.log(risk);
                    return "GOLD";
          };
          this.type = findType();
}

var customer = new Customer("ABC Customer"); // ABC Customer
                                                  // 0
console.log(customer.type); // GOLD
console.log(customer.risk); // undefined
```

If a private inner function is returned outside, its methods can be called from outside:

```
function Outer(){
          return new Inner();

          //private inner
          function Inner(){
                    this.sayHello = function(){
                              console.log("Hello");
                    }
          }
}

(new Outer()).sayHello(); // Hello
```

Privileged Methods

Prototype methods can't access private fields; everything attached to them is also "public". We need a way to declare methods so that (a) they are accessible publicly and (b) they can access private members, which means privileged or protected access. For this purpose, the following code can be used:

```
function Customer(orderAmount){
          // private field
          var cost = orderAmount / 2;
          this.orderAmount = orderAmount;
          var that = this;

          // privileged method
          this.calculateProfit = function(){
                    return that.orderAmount - cost;
          };
}

Customer.prototype.report = function(){
          console.log(this.calculateProfit());
};

var customer = new Customer(3000);
customer.report(); // 1500
```

Public Fields

For public fields, prototype or instance can be used. Prototype fields and methods are shared by new instances. (Prototype objects are also shared). If a new instance changes the field or method in its object, the change is not reflected in other instances. To change all instances, we need to change it in the prototype object.

```
function Customer(name,orderAmount){
          // public fields
          this.name = name;
          this.orderAmount = orderAmount;
}
Customer.prototype.type = "NORMAL";

Customer.prototype.report = function(){
          console.log(this.name);
          console.log(this.orderAmount);
          console.log(this.type);
          console.log(this.country);
};

Customer.prototype.promoteType = function(){
          this.type = "SILVER";
};
```

(Code snippet runs to next page.) ---------->

DZone, Inc. | www.dzone.com

```
var customer1 = new Customer("Customer 1",10);
// public field
customer1.country = "A Country";
customer1.report(); // Customer 1
                    // 10
                    // NORMAL
                    // A Country

var customer2 = new Customer("Customer 2",20);
customer2.promoteType();
console.log(customer2.type); // SILVER
console.log(customer1.type); // NORMAL
```

Public Methods

Prototype methods are public. Any methods attached to "this" are also public.

```
function Customer(){
        // public method
        this.shipOrder = function(shipAmount){
                return shipAmount;
        };
}
// public method
Customer.prototype.addOrder = function (orderAmount) {
        var totalOrder = 0;
        for(var i = 0; i < arguments.length; i++) {
                totalOrder += arguments[i];
        }
        return totalOrder;
};

var customer = new Customer();
// public method
customer.findType = function(){
        return "NORMAL";
};
console.log(customer.addOrder(25,75)); // 100
console.log(customer.shipOrder(50)); // 50
console.log(customer.findType()); // NORMAL
```

INHERITANCE

There are different ways to implement inheritance in JavaScript. "Prototypal Inheritance" – using prototype as an inheritance mechanism – has many advantages. For example:

```
function Parent(){
        var parentPrivate = "parent private data";
        var that = this;
        this.parentMethodForPrivate = function(){
                return parentPrivate;
        };
        console.log("parent");
}
Parent.prototype = {
        parentData: "parent data",
        parentMethod: function(arg){
                return "parent method";
        },
        overrideMethod: function(arg){
                return arg + " overriden parent method";
        }
}

function Child(){
// super constructor is not called, we have to invoke it
        Parent.call(this);
        console.log(this.parentData);
        var that = this;
        this.parentPrivate = function(){
                return that.parentMethodForPrivate();
        };
        console.log("child");
}

//inheritance
Child.prototype = new Parent();// parent
Child.prototype.constructor = Child;

//lets add extented functions
Child.prototype.extensionMethod = function(){
        return "child's " + this.parentData;
};
```

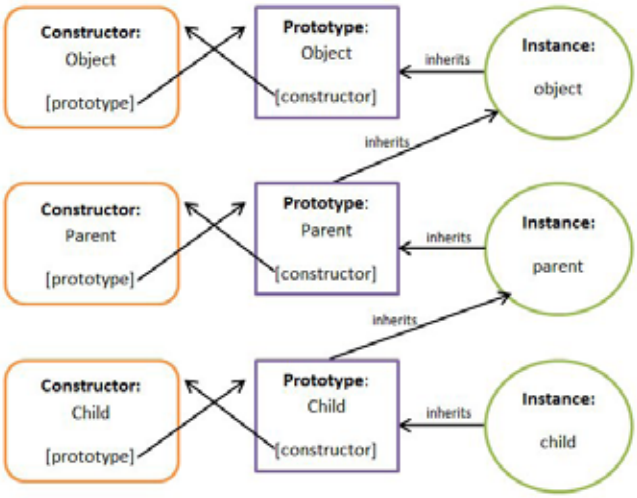
(Code snippet runs to next column.) ---------->

```
//override inherited functions
Child.prototype.overrideMethod = function(){
        //parent's method is called
        return "Invoking from child" + Parent.prototype.
overrideMethod.call(this, " test");
};

var child = new Child();// parent
                // parent data
                // child
console.log(child.extensionMethod()); //child's parent data
console.log(child.parentData); //parent data
console.log(child.parentMethod()); //parent method
console.log(child.overrideMethod()); //Invoking from child test
overriden parent method
console.log(child.parentPrivate()); // parent private data

console.log(child instanceof Parent); //true
console.log(child instanceof Child); //true
```

When a member of child object is accessed, it is searched from the object's members. If it is not found there, then it is searched in its prototype object. If it is not found there, then it is searched in the parent object's members and prototype. It is searched up to the Object's prototype. This hierarchy is also called the "Prototype chain". The following diagram illustrates the chain:



MODULARIZATION

Once we have many custom object types, we need to group them and manage their visibility, dependencies, and loading. Namespace and Modules can be used to handle these tasks. (Note that EcmaScript version 6 (Harmony) is beginning to develop a full-fledged module system. Until it is implemented by browsers, there are some useful libraries to facilitate these features.)

Namespaces

There is no direct namespace support in JavaScript. We can create namespace via objects and fill our object types in object properties:

```
//create namespace
var myLib = {};
myLib.myPackage = {};

//Register types to namespace
myLib.myPackage.MyType1 = MyType1;
myLib.myPackage.MyType2 = MyType2;
```

Modules

Modules are a way of dividing our JavaScript code-base into packages. Modules export some object type definitions and import other modules. The module system loads each module and its dependencies. First it runs the dependencies in appropriate order; then the module code is executed. Some libraries can be used for this purpose.

In modules, we define new types, import some types from other modules and export our public types. Here is an example of a module definition:

```
Module.define("module1.js",["dependent_module1.js","dependent_
module2.js",...], function(dependentMod1, dependentMod2) { //
IMPORTS

//TYPE DEFINITIONS
function ExportedType1(){
          // use of dependent module's types
          var dependentType = new dependentMod1.DependentType1();
          ...
}
function ExportedType2(){
}
...

// EXPORTS
return { ExportedType1: ExportedType1, ExportedType2:
ExportedType2,...};

});

To use a module (can work asynchronously or synchronously):
Module.use(["module1.js"], function(aModule){
          console.log("Loaded aModule!");
          var AType = aModule.AnExportedType;
          var atype1Instance = new AType();
});
```

CUSTOM EXCEPTIONS

In JavaScript, there are built-in exceptions like Error, TypeError and SyntaxError which may be created and thrown during runtime. Every exception is "unchecked". An ordinary object can be used as an exception with throw statement. Thus, we can create our own custom exceptions and throw and catch them. It is good practice to write exceptions that extend JavaScript's standard Error object.

```
function BaseException() {
}
BaseException.prototype = new Error();
BaseException.prototype.constructor = BaseException;

BaseException.prototype.toString = function(){
          // note that name and message are properties of Error
          return this.name + ": " + this.message;
};

function NegativeNumberException(value) {
          this.name = "NegativeNumberException";
          this.message = "Negative number! Value: " + value;
}
NegativeNumberException.prototype = new BaseException();
NegativeNumberException.prototype.constructor =
NegativeNumberException;

function EmptyInputException() {
          this.name = "EmptyInputException";
          this.message = "Empty input!";
}
EmptyInputException.prototype = new BaseException();
EmptyInputException.prototype.constructor = EmptyInputException;

var InputValidator = (function(){

var InputValidator = {};
InputValidator.validate = function(data){
          var validations = [validateNotNegative,validateNotEmp
ty];
          for(var i = 0; i < validations.length; i++){
                    try {
                              validations[i](data);
                    }
                    catch (e) {
                              if (e instanceof
NegativeNumberException) {
                                        //re-throw
                                        throw e;
                              }
                              else if (e instanceof
EmptyInputException) {
                                        // tolerate it
                                        data = "0";
                              }
                    }
          }
};
return InputValidator;

function validateNotNegative(data){
                    if (data < 0)
                              throw new NegativeNumberException(data)
function validateNotEmpty(data)
{
                    if (data == "" || data.trim() == "")
                              throw new EmptyInputException();
}

})();

try{
          InputValidator.validate("-1");
}
catch(e){
          console.log(e.toString()); // NegativeNumberException:N
egative number! Value: -1

console.log("Validation is done."); // Validation is done.

var validations = [validateNotNegative,validateNotEmpty];
}
```

(Code snippet runs to next page.) ---------->

CUSTOM EVENTS

Custom events reduce code complexity and decrease coupling of objects. Here is a typical event pattern:

```
function EventManager(){
}
var listeners = {};

EventManager.fireEvent = function(eventName, eventProperties) {
          if (!listeners[eventName])
                    return;

          for (var i = 0; i < listeners[eventName].length; i++) {
                    listeners[eventName][i](eventProperties);
          }
};

EventManager.addListener = function(eventName, callback) {
          if (!listeners[eventName])
                    listeners[eventName] = [];
          listeners[eventName].push(callback);
};

EventManager.removeListener = function(eventName, callback) {
          if (!listeners[eventName])
                    return;
          for (var i = 0; i < listeners[eventName].length; i++) {
                    if (listeners[eventName][i] == callback) {
                              delete listeners[eventName][i];
                              return;
                    }
          }
};

EventManager.addListener("popupSelected", function(props){
          console.log("Invoked popupSelected event: " + props.
itemID);
});

EventManager.fireEvent("popupSelected", {itemID:"100"}); //
Invoked popupSelected event: 100
```

WRITING COMPONENTS

JavaScript lets developers add new behaviors to HTML elements. These enrichments can be combined in a component structure. There is a continuing specification work by the W3C on "Web Components". Here is a commonly used pattern to bind JavaScript objects to DOM objects. In this way, we can collect behaviors and manage their lifecycles.

1- Define JavaScript Component:

```
function InputTextNumberComponent(domElement){
          this.initialize(domElement);
}

InputTextNumberComponent.prototype.initialize =
function(domElement){
          domElement.onchange = function(){
                    //just a format
                    domElement.value = "-" + domElement.value +
"-";
          };
domElement.jsComponent = this; //Expando property
          this.domElement = domElement;
};

InputTextNumberComponent.prototype.resetValue = function(){
          this.domElement.value = "";
};
```

2- Define CSS Class to Correlate HTML Elements with JavaScript Components:

```
<style type="text/css">
.inputTextNumber { text-align:right; }
</style>
```

HTML element should be as follows:

```
<input type="text" class="inputTextNumber" name="NumberField"
size="10" value="Click me!" onClick="this.jsComponent.
resetValue()">
```

3- When page is loaded (or DOM ready), detect HTML elements and create components:

```
window.onload = function(){
          var inputTextNumbers = document.getElementsByClassName(
"inputTextNumber");
          for(var i = 0; i < inputTextNumbers.length; i++){
                    var myComp = new InputTextNumberComponent(
inputTextNumbers.item(i));
          }
};
```

ABOUT THE AUTHORS

Ibrahim Levent has worked as ERP architect and project leader for 12 years at a manufacturing group of companies in Turkey. His team developed an ERP system with Java EE and web technologies. Recently he has been managing abroad ERP projects and dealing with localization issues. In his spare time, he likes to write about software architecture and programming in his blog.

RECOMMENDED BOOK

This book is a solid reference for object-oriented programming with JavaScript. At the beginning of the book, it explains basic concepts of object-oriented JavaScript programming. Then it covers important desing patterns with detailed and clean examples.

BUY HERE

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, blogs, feature articles, source code and more. "DZone is a developer's dream," says PC Magazine.

DZone, Inc.
150 Preston Executive Dr.
Suite 201
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-936502-72-1
ISBN-10: 1-936502-72-0
50795
9 781936 502721

$7.95

Version 1.0