OOP In JavaScript: What You NEED to Know

MARCH. 19 2013 216

(Object Oriented JavaScript: Only Two Techniques Matter)

menu

Prerequisite:
JavaScript Objects in Detail
JavaScript Prototype

Object Oriented Programming (OOP) refers to using self-contained pieces of code to develop applications. We call these self-contained pieces of code **objects**, better known as *Classes* in most OOP programming languages and *Functions* in JavaScript. We use objects as building blocks for our applications. Building applications with objects allows us to adopt some valuable techniques, namely, **Inheritance** (objects can inherit features from other objects), **Polymorphism** (objects can share the same interface—how they are accessed and used—while their underlying implementation of the interface may differ), and **Encapsulation** (each object is responsible for specific tasks).

In this article, we are concerned with only Inheritance and Encapsulation since only these two concepts apply to OOP in JavaScript, particularly because, in JavaScript, objects can encapsulate functionalities and inherit methods and properties from other objects. Accordingly, in the rest of the article, I discuss everything you need to know about using objects in JavaScript in an object oriented manner—with inheritance and encapsulation—to easily reuse code and abstract functionalities into specialized objects.

Table of Contents

- Receive Updates
- Encapsulation and Inheritance Overview
- OOP in JavaScript
- Encapsulation in JavaScript
- Inheritance in JavaScript
- Implementing the Parasitic Combination Inheritance Pattern
- Final Words

Receive Updates

We will focus **on only the best two techniques**[1] for implementing OOP in JavaScript. Indeed, many techniques exist for implementing OOP in JavaScript, but rather than evaluate each, I choose to focus on the two best techniques: the best technique for creating objects with specialized functionalities (aka Encapsulation) and the best technique for reusing code (aka Inheritance). By "best" I mean the most apt, the most efficient, the most robust.

Encapsulation and Inheritance Overview

Objects can be thought of as the main actors in an application, or simply the main "things" or building blocks that do all the work. As you know by now, objects are everywhere in JavaScript since every component in JavaScript is an Object, including Functions, Strings, and Numbers. We normally use object literals or constructor functions to create objects.

Encapsulation refers to enclosing all the functionalities of an object within that object so that the object's internal workings (its methods and properties) are hidden from the rest of the application. This allows us to abstract or localize specific set of functionalities on objects.

Inheritance refers to an object being able to inherit methods and properties from a parent object (a Class in other OOP languages, or a Function in JavaScript).

Both of these concepts, encapsulation and inheritance, are important because they allow us to build applications with reusable code, scalable architecture, and abstracted functionalities. Maintainable, scalable, efficient.

An **instance** is an implementation of a Function. In simple terms, it is a copy (or "child") of a Function or object. For example:

```
1 // Tree is a constructor function because we will use new keyword to invoke it.
2 function Tree (typeOfTree) {}
3
4 // bananaTree is an instance of Tree.
5 var bananaTree = new Tree ("banana");
```

In the preceding example, *bananaTree* is an object that was created from the Tree constructor function. We say that the *bananaTree* object is an instance of the Tree object. Tree is both an object and a function, because functions are objects in JavaScript. *bananaTree* can have its own methods and properties and inherit methods and properties from the Tree object, as we will discuss in detail when we study inheritance below.

OOP in JavaScript

The two important principles with OOP in JavaScript are Object Creation patterns (**Encapsulation**) and Code Reuse patterns (**Inheritance**). When building applications, you create many objects, and there exist many ways for creating these objects: you can use the ubiquitous object literal pattern, for example:

```
1 var myObj = {name: "Richard", profession: "Developer"};
```

You can use the prototype pattern, adding each method and property directly on the object's prototype. For example:

```
1 function Employee () {}
2
```

```
Employee.prototype.firstName = "Abhijit";
Employee.prototype.lastName = "Patel";
Employee.prototype.startDate = new Date();
Employee.prototype.signedNDA = true;
Employee.prototype.fullName = function () {
console.log (this.firstName + " " + this.lastName);
};

var abhijit = new Employee () //
console.log(abhijit.fullName()); // Abhijit Patel
console.log(abhijit.signedNDA); // true
```

You can also use the constructor pattern, a constructor function (Classes in other languages, but Functions in JavaScript). For example:

```
function Employee (name, profession) {
this.name = name;
this.profession = profession;
} // Employee () is the constructor function because we use the <em>new</em> keyword below to invo

var richard = new Employee ("Richard", "Developer") // richard is a new object we create from the Emplo

console.log(richard.name); //richard
console.log(richard.profession); // Developer
```

In the latter example, we use a custom constructor function to create an object. This is how we create objects when we want to add methods and properties on our objects, and when we want to encapsulate functionality on our objects. JavaScript developers have invented many patterns (or ways) for creating objects with constructor functions. And when we say Object Creation Patterns, we are concerned principally with the many ways of creating objects from constructor functions, as in the preceding example.

In addition to the patterns for creating objects, you want to reuse code efficiently. When you create your objects, you will likely want some of them to inherit (have similar functionality) methods and properties from a parent object, yet they should also have their own methods and properties. Code reuse patterns facilitate ways in which we can implement inheritance.

These two universal principles—creating objects (especially from constructor Functions) and allowing objects to inherit properties and methods—are the main focus of this article and, indeed, the main concepts with OOP in JavaScript. We first discuss the object creation pattern.

Encapsulation in JavaScript

(The Best Object Creation Pattern: Combination Constructor/Prototype Pattern)

As discussed above, one of the main principles with OOP is encapsulation: put all the inner workings of an object inside that object. To implement encapsulation in JavaScript, we have to define the core methods and properties on that object. To do this, we will use the best pattern for encapsulation in **JavaScript: the Combination Constructor/Prototype Pattern**. This name is a mouthful, but you needn't memorize it, since we are only concerned with its implementation. Before we implement it, let's quickly learn a bit more about the practicality of encapsulation.

Why Encapsulation?
When you simply want to create an object just to store some data, and it is the only object of its kind, you can use an object literal and create your object. This is quite common and you will use this simple pattern often.

However, whenever you want to create objects with similar functionalities (to use the same methods and properties), you encapsulate the main functionalities in a Function and you use that Function's constructor to create the objects. This is the essence of encapsulation. And it is this need for encapsulation that we are concerned with and why we are using the Combination Constructor/Prototype Pattern.

To make practical use of OOP in JavaScript, we will build an object-oriented quiz application that uses all the principles and techniques we learn in this article. First up, our quiz application will have users (a **Users** Function) who take the quiz. There will be some common properties for every user who takes the quiz: each user will have a *name*, a *score*, an *email*, and the *quiz scores* (all the scores). These are the properties of the User object. In addition, each User object should be able to show the name and score, save scores, and change the email. These are the methods of the object.

Because we want ALL the user objects to have these same properties and methods, we cannot use the object literal way of creating objects. We have to use a constructor Function to

encapsulate these properties and methods.

Since we know all users will have the same set of properties, it makes sense to create a Function (Class in OOP languages) that encapsulates these properties and methods. Thus, we will use the Combination Constructor/Prototype Pattern for this.

Implementation of Combination Constructor/Prototype Pattern

The User Function:

I will explain each line.

```
function User (theName, theEmail) {
    this.name = theName;
    this.email = theEmail;
    this.quizScores = [];
    this.currentScore = 0;
}

User.prototype = {
    constructor: User,
    saveScore:function (theScoreToAdd)  {
        this.quizScores.push(theScoreToAdd)
    },
    showNameAndScores:function ()  {
        var scores = this.quizScores.length > 0 ? this.quizScores.join(",") : "No Scores Yet";
        return this.name + " Scores: " + scores;
    },
    changeEmail:function (newEmail)  {
        this.email = newEmail;
        return "New Email Saved: " + this.email;
    }
}
```

Make Instances of the User function

```
// A User
firstUser = new User("Richard", "Richard@examnple.com");
firstUser.changeEmail("RichardB@examnple.com");
```

```
firstUser.saveScore(15);
firstUser.saveScore(10);

firstUser.showNameAndScores(); //Richard Scores: 15,10

// Another User
secondUser = new User("Peter", "Peter@examnple.com");
secondUser.saveScore(18);
secondUser.showNameAndScores(); //Peter Scores: 18
```

Explanation of Combination Constructor/Prototype Pattern

Let's expound on each line of code so we have a thorough understanding of this pattern.

The following lines initialize the instance properties. These properties will be defined on each User instance that is created. So the values will be different for each user. The use of the this keyword inside the function specifies that these properties will be unique to every instance of the User object:

```
this.name = theName;
this.email = theEmail;
this.quizScores = [];
this.currentScore = 0;
```

In the code below, we are overwriting the prototype property with an object literal, and we define all of our methods (that will be inherited by all the User instances) in this object. Discussion continues after the code:

```
User.prototype = {
    constructor: User,
    saveScore:function (theScoreToAdd)  {
        this.quizScores.push(theScoreToAdd)
    },
    showNameAndScores:function ()  {
        var scores = this.quizScores.length > 0 ? this.quizScores.join(",") : "No Scores Yet";
        return this.name + " Scores: " + scores;
    },
    changeEmail:function (newEmail)  {
        this.email = newEmail;
```

```
12          return "New Email Saved: " + this.email;
13      }
14  }
```

This way of overwriting the constructor is simply for convenience, so we don't have to write User.prototype each time, like this:

```
1  User.prototype.constructor = User;
2  User.prototype.saveScore = function (theScoreToAdd)  {
3      this.quizScores.push(theScoreToAdd)
4  };
5
6  User.prototype.showNameAndScores = function ()  {
7      var scores = this.quizScores.length > 0 ? this.quizScores.join(",") : "No Scores Yet";
8      return this.name + " Scores: " + scores;
9  };
10
11 User.prototype.changeEmail =  function (newEmail)  {
12     this.email = newEmail;
13     return "New Email Saved: " + this.email;
14 }
```

By overwriting the prototype with a new object literal we have all the methods organized in one place, and you can better see the encapsulation that we are after. And of course it is less code you have to type.

JavaScript Prototype

In JavaScript, you add methods and properties on the prototype property when you want instances of an object to inherit those methods and properties. This is the reason we add the methods on the User.prototype property, so that they can be used by all instances of the User object. Read more about JavaScript Prototype in Plain Language.

Constructor Property

In my post JavaScript Prototype, I explained that every function has a constructor property, and this property points to the constructor of the function. For example:

```
1  function Fruit () {}
```

```
var newFruit = new Fruit ();
console.log (newFruit.constructor) // Fruit ()
```

The one disadvantage of overwriting the prototype is that the constructor property no longer points to the prototype, so we have to set it manually. Hence this line:

```
constructor: User
```

Prototype Methods

In the following lines, we create methods on the prototype (in the object literal) so that all instances of Users can have access to these methods.

```
saveScore:function (theScoreToAdd)  {
        this.quizScores.push(theScoreToAdd)
    },
    showNameAndScores:function ()  {
        var scores = this.quizScores.length > 0 ? this.quizScores.join(",") : "No Scores Yet";
        return this.name + " Scores: " + scores;
    },
    changeEmail:function (newEmail)  {
        this.email = newEmail;
        return "New Email Saved: " + this.email;
    }
```

We then created instances of the User object:

```
// A User
firstUser = new User("Richard", "Richard@examnple.com");
firstUser.changeEmail("RichardB@examnple.com");
firstUser.saveScore(15);
firstUser.saveScore(10);

firstUser.showNameAndScores(); //Richard Scores: 15,10

// Another User
secondUser = new User("Peter", "Peter@examnple.com");
```

```
secondUser.saveScore(18);
secondUser.showNameAndScores(); //Peter Scores: 18
```

As you see, we have encapsulated all the functionality for a User inside the User Function, so that each instance of User can make use of the prototype methods (like changeEmail) and define their own instance properties (like name and email).

With this pattern, you can use the standard operators and methods on the instances, including the instanceOf operator, the for-in loop (even hasOwnProperty), and the constructor property.

Inheritance in JavaScript

(The Best Pattern: Parasitic Combination Inheritance)

Implementing inheritance in our quiz application will permit us to inherit functionality from parent Functions so that we can easily reuse code in our application and extend the functionality of objects. Objects can make use of their inherited functionalities and still have their own specialized functionalities.

The best pattern for implementing inheritance in JavaScript is the Parasitic Combination inheritance [2]. Before we dive into this awesome pattern, let's see why its practical to use inheritance in our applications.

We have successfully implemented encapsulation by enclosing all the functionality for users of our quiz application by adding all the methods and properties that each user will need on the User function, and all instances of User will have those properties and methods.

Why Inheritance?

Next, we want to encapsulate all the functionalities for every Question. The Question function (Class in OOP languages) will have all the generic properties and methods that every kind of question will need to have. For example, every question will have the question, the choices, and the correct answer. These will be the properties. In addition, each question will have some methods: getCorrectAnswer and getUserAnswer, and displayQuestion.

We want our quiz application to make different types of Questions. We will implement a MultipleChoiceQuestion function and a DragDropQuestion function. To implement these

would not make sense to put the properties and methods outlined above (that all questions will use) inside the MultipleChoiceQuestion and DragDropQuestion functions separately, repeating the same code. This would be redundant.

Instead, we will leave those properties and methods (that all questions will use) inside the Question object and make the MultipleChoiceQuestion and DragDropQuestion functions inherit those methods and properties. This is where inheritance is important: we can reuse code throughout our application effectively and better maintain our code.

Since the MultipleChoiceQuestion HTML layout and will be different from the DragDropQuestion HTML layout, the displayQuestion method will be implemented differently in each. So we will override the displayQuestion method on the DragDropQuestion. Overriding functions is another principle of OOP.

Lets Code.

Implementing the Parasitic Combination Inheritance Pattern

To implement this pattern, we have to use two techniques that were invented specifically for inheritance in JavaScript. Some notes about these techniques follow. No need to memorize any of the detail; just understand and be aware of the techniques.

Prototypal Inheritance by Douglas Crockford

Douglas Crockford created the following Object.create method [3], used in a fundamental way to implementing inheritance with the pattern we are using.

Object.create method

Ruminate on the method Crockford created:

```
1  if (typeof Object.create !== 'function') {
2      Object.create = function (o) {
3          function F() {
4          }
5
6          F.prototype = o;
7          return new F();
```

```
    };
}
```

This method has been added to the ECMAScript5 specification, and you can access it with Object.create (). Let's quickly understand it is doing.

```
Object.create = function (o) {
//It creates a temporary constructor F()
        function F() {
        }
//And set the prototype of the this constructor to the parametric (passed-in) o object
//so that the F() constructor now inherits all the properties and methods of o
        F.prototype = o;

//Then it returns a new, empty object (an instance of F())
//Note that this instance of F inherits from the passed-in (parametric object) o object.
//Or you can say it copied all of the o object's properties and methods
        return new F();
    }
```

The crux of the matter with this Object.create method is that you pass into it an object that you want to inherit from, and it returns a new object that inherits from the object you passed into it. For example:

```
// We have a simple cars object
var cars = {
    type:"sedan",
    wheels:4
};

// We want to inherit from the cars object, so we do:
var toyota = Object.create (cars); // now toyota inherits the properties from cars
console.log(toyota.type); // sedan
```

Of course we can now add more properties to the toyota object, but let's move on.

The next function we will use for inheritance is the **inheritPrototype** function. This function succinctly implements the parasitic combination inheritance for us. We pass in the parent

(or Super Class) and the child object (or Sub Class), and the function does the parasitic combination inheritance: makes the child object inherits from the parent object.

```
function inheritPrototype(childObject, parentObject) {
  // As discussed above, we use the Crockford's method to copy the properties and methods from the
// So the copyOfParent object now has everything the parentObject has
  var copyOfParent = Object.create(parentObject.prototype);

  //Then we set the constructor of this new object to point to the childObject.
// Why do we manually set the copyOfParent constructor here, see the explanation immediately follow
  copyOfParent.constructor = childObject;

  // Then we set the childObject prototype to copyOfParent, so that the childObject can in turn inherit
  childObject.prototype = copyOfParent;
}
```

Why did we manually set the copyOfParent.constructor?

_____

We explicitly set the *copyOfParent.constructor* property to point to the childObject constructor because in the preceding step, var copyOfParent = Object.create(parentObject.prototype), this is what we actually did:

```
// We made a new object and overwrote its prototype with the parentObject prototype:
function F() {
      }
F.prototype = parentObject.prototype;
// Then it was this new F object we assigned to copyOfParent.
// All of this was done inside the Object.create () method.
```

So, this new F object, which we assigned to copyOfParent, doesn't have a constructor property anymore because we overwrote its entire prototype. Whenever you overwrite an object's prototype (object.prototype = someVal), you also overwrite the object's constructor property.

To make sure we have the correct value for copyOfParent constructor, we set it manually with this:
copyOfParent.constructor = childObject;

A commenter by the name of John correctly pointed our that I did not corruptly explain this bit, hence this detailed explanation.

_____

Essentially, we are copying all the properties and methods from the parentObject to the childObject, but we are using the copyOfParent as an intermediary for the copy. And because the childObject prototype was overwritten during the copy, we manually set the copyOfParent constructor to the childObject. Then we set the childObject prototype to the copyOfParent so that the childObject inherits from the parentObject.

Okay, that was quite a bit. I am hopeful you understand some of that :).

Back to the fun stuff: Creating our quiz OOP style.
Now that we understand the inheritPrototype function we will be using, lets go ahead and implement our Question constructor.

Note that I use "constructor" and "function" interchangeably sometimes in this particular article when referring to the function, because the function will be used as a constructor to create instances.

The Question Constructor (Parent of all Question Objects):
(Can be thought of as the Super Class for Questions)

```
1  // The Question function is the parent for all other question objects;
2  // All question objects will inherit from this Question constructor
3
4  function Question(theQuestion, theChoices, theCorrectAnswer) {
5      // Initialize the instance properties
6      this.question = theQuestion;
7      this.choices = theChoices;
8      this.correctAnswer = theCorrectAnswer;
9      this.userAnswer = "";
10
11     // private properties: these cannot be changed by instances
12     var newDate = new Date(),
13     // Constant variable: available to all instances through the instance method below. This is also a priva
14         QUIZ_CREATED_DATE = newDate.toLocaleDateString();
15
```

```
// This is the only way to access the private QUIZ_CREATED_DATE variable
// This is an example of a privilege method: it can access private properties and it can be called publicly
    this.getQuizDate = function () {
        return QUIZ_CREATED_DATE;
    };

// A confirmation message that the question was created
    console.log("Quiz Created On: " + this.getQuizDate());

}
```

Add Prototype Methods to The Question Object

All instances of the Question object will inherit these methods, because we are adding the methods on the Question prototype.

```
 // Define the prototype methods that will be inherited
Question.prototype.getCorrectAnswer = function () {
    return  this.correctAnswer;
};

Question.prototype.getUserAnswer = function () {
    return this.userAnswer;
};

Question.prototype.displayQuestion = function () {
    var questionToDisplay = "<div class='question'>" + this.question + "</div><ul>";
        choiceCounter = 0;

    this.choices.forEach(function (eachChoice)  {
        questionToDisplay += '<li><input type="radio" name="choice" value="' + choiceCounter + '">' +
        choiceCounter++;
    });
    questionToDisplay += "</ul>";

    console.log (questionToDisplay);
};
```

Child Questions (Sub Classes of the Question object)

Now that we have the Question constructor object setup, we can inherit from it and create sub classes (children objects). The power of inheritance is that we can create all sorts of questions now, and each can be quite versatile.

First, a **Multiple Choice Question:**

```
1  // Create the MultipleChoiceQuestion
2  function MultipleChoiceQuestion(theQuestion, theChoices, theCorrectAnswer){
3  // For MultipleChoiceQuestion to properly inherit from Question, here inside the MultipleChoiceQuestio
4  // passing MultipleChoiceQuestion as the this object, and the parameters we want to use in the Questio
5      Question.call(this, theQuestion, theChoices, theCorrectAnswer);
6  };
```

And then we have to use the inheritPrototype function we discussed moments ago:

```
1  // inherit the methods and properties from Question
2  inheritPrototype(MultipleChoiceQuestion, Question);
```

After we have inherited from Question, we then add methods to the MultipleChoiceQuestion function, if necessary. But we must do it after we inherit, not before, or all the methods we define on its prototype will be overwritten. We are not adding any now.

A Drag and Drop Question

In a similar manner, we can make yet another type of question:

```
1  // Create the DragDropQuestion
2  function DragDropQuestion(theQuestion, theChoices, theCorrectAnswer) {
3      Question.call(this, theQuestion, theChoices, theCorrectAnswer);
4  }
5
6  // inherit the methods and properties from Question
7  inheritPrototype(DragDropQuestion, Question);
```

Overriding Methods

Overriding methods is a another principle of OOP, and we can do it easily with this pattern. Since the Drag and Drop questions will have a different HTML layout from the Multiple C

questions (no radio buttons, for example), we can override the displayQuestion method so it operates specifically to the Drag and Drop question needs:

```
// Override the displayQuestion method it inherited
DragDropQuestion.prototype.displayQuestion = function () {
    // Just return the question. Drag and Drop implementation detail is beyond this article
    console.log(this.question);
};
```

In our real Quiz application, we would create a Quiz constructor that is the main application that launches the quiz, but in this article, we can test our inheritance code by simply doing this:

```
// Initialize some questions and add them to an array
var allQuestions = [
new MultipleChoiceQuestion("Who is Prime Minister of England?", ["Obama", "Blair", "Brown", "Camerc

new MultipleChoiceQuestion("What is the Capital of Brazil?", ["São Paulo", "Rio de Janeiro", "Brasília"], 2

new DragDropQuestion("Drag the correct City to the world map.", ["Washington, DC", "Rio de Janeiro",
];

// Display all the questions
allQuestions.forEach(function (eachQuestion)  {
    eachQuestion.displayQuestion();
});
```

If you run the code, you will see that the displayQuestion for the multiple choice questions returns the **question** in a div tag, with **choices** formatted with radio buttons inside li tags. On the other hand, the drag and drop questions displayQuestion method simply returns the **question** without the choices.

Nicholas Zakas stated it wonderfully, “Parasitic combination inheritance is considered the most optimal inheritance paradigm” [5] in JavaScript. If you learn it and understand it well, you should use it in your JavaScript web applications.

You might be wondering how is the Combination Constructor/Prototype Pattern we used for Encapsulation earlier different from the Parasitic Combination Inheritance. They are simila

the former is best used for encapsulation (creating custom objects), and it does not have all the inheritance mechanisms such as subclassing (creating child constructors that inherit from the parent constructor). Moreover, the inheritance pattern goes beyond setting up objects to just inherit properties and methods, it enables child objects to themselves be parent objects of other objects, and you can use private members, overriding, and other OOP concepts.

Final Words

I gave you the full details for implementing the best two patterns for OOP in JavaScript, and I am hopeful you understood at least the general concepts. Go use these patterns in your JavaScript applications. Note that you can use OOP in even small and medium applications, not just complex applications.

Be good and do good work, and take care of yourself: sleep well, eat well, and enjoy life.

Further Reading

Read chapters 6 and 7 of Zakas's *Professional JavaScript for Web Developers*.
Read chapters 5 and 6 of Stefanov's *JavaScript Pattens*.
Read chapter 4 of Herman's *Effective JavaScript*.

Notes

1. Professional JavaScript for Web Developers Chapter 6.
2. JavaScript Patterns Chapters 5 and 6
3. Effective JavaScript: 68 Specific Ways to Harness the Power of JavaScript Chapter 4.
4. http://javascript.crockford.com/prototypal.html
5. Professional JavaScript for Web Developers Chapter 6, page 215.

Posted in: Advanced javaScript, JavaScript / Tagged: Advanced JavaScript, Learn JavaScript, OOP JavaScript

Richard

Thanks for your time; please come back soon. Email me here: javascriptissexy at gmail email, or use the conta

216 Comments

Anthony

March 19, 2013 at 11:33 pm / Reply

I was just doing the same-thing tonight before I found your article!

```
var config = {
writable: true,
enumerable: true,
configurable: true
}

var defineProperty = function(obj, name, value) {
config.value = value;
Object.defineProperty(obj, name, config);
}

var man = Object.create(null);
defineProperty(man, 'sex', 'male');

var tony = Object.create(man);
defineProperty(tony, 'firstName', 'Tony');
defineProperty(tony, 'lastName', 'Brown');

alert.log(tony.sex);
alert(tony.firstName);
alert(tony.lastName);

console.log(Object.getPrototypeOf(tony));
```

Richard Bovell (Author)

March 20, 2013 at 1:01 am / Reply

I am glad to see that you are practicing this stuff. Way to go, Tony

Boom

July 9, 2013 at 12:28 pm / Reply

No

Anthony

March 19, 2013 at 11:40 pm / Reply

Excellent article, as usual Richard! I commend you for your work and promotion of the JavaScript language!

Richard Bovell (Author)

March 20, 2013 at 1:05 am / Reply

Thanks, Man. I appreciate it. Some of these articles take quite a bit of time to write. I always think I will finish in some length of time, but the articles more often than not seem to take twice as long.

Anthony

March 20, 2013 at 4:12 am / Reply

Keep it up Richard, I'll be singing your praises on Twitter and such!

Thanawat

March 20, 2013 at 6:46 am / Reply

Another great article, Richard.

One thing, in the snippet function inheritPrototype
Should
var copyOfParent = Object.create(superType.prototype);

be
var copyOfParent = Object.create(parentObject.prototype);

Richard Bovell (Author)

March 20, 2013 at 10:10 am / Reply

Good catch, Thanawat. I have fixed it.

Jurgen

March 20, 2013 at 9:04 am / Reply

Thank you for these wonderful articles. I have read through your previous articles on objects and prototype, but I can't seem to understand the real difference (if any) between the following two pieces of code:

```
function Test (theVariable) {
this.variable = theVariable;
this.variable2 = 'Some text ';
this.someMethod = function () {
console.log(this.variable2 + this.variable);
}
}
```

and:

```
function Test (theVariable) {
this.variable = theVariable;
}
test.prototype = {
constructor : Test,
variable2 : 'Some text ';
someMethod : function () {
console.log(this.variable2 + this.variable);
}
}
```

Thanks a lot!

Richard Bovell (Author)

March 20, 2013 at 11:01 am / Reply

Good question, Jurgen.

The two examples will function exactly the same with your test code. But they are fundamentally different.

In the former, all the properties and methods are defined on the instance, so the values will be unique to each instance. You can see the issue better if we use a reference type (Array or Objects). I changed your code and made variable2 an array.

```
1
2  // Constructor Pattern
3  function Test(theVariable) {
4      this.variable = theVariable;
5      this.variable2 = ["Mike", "Anil"];
6      this.someMethod = function () {
7          console.log(this.variable2 + this.variable);
8      }
9  }
10
11 var aTest = new Test ("-- Testing");
12 aTest.variable2.push("Richard");
13 aTest.someMethod(); // Mike,Anil,Richard -- Testing
14
15 var anotherTest = new Test ("-- Testing");
16 anotherTest.variable2.push("Jurgen");
17 // Array is unique to the anotherTest instance
18 anotherTest.someMethod(); //Mike,Anil,Jurgen-- Testing
19
20 // variable2 was not changed by the anotherTest instance:
21 aTest.someMethod(); // Mike,Anil,Richard-- Testing
22
```

In your latter example, however, only this.variable is an instance variable, so it will be unique always. All the properties and methods defined on the prototype will be inherited by all instances of Test, and all instances will have the same values (for reference types) for those. For example:

```
1
2  function Test(theVariable) {
3      this.variable = theVariable;
4  }
5
6  Test.prototype = {
7      constructor:Test,
8      variable2:["Mike", "Anil"],
9      someMethod:function () {
10         console.log(this.variable2 + " " +  this.variable);
11     }
12 }
13
14
15 var aTest = new Test ("-- Testing");
16 aTest.variable2.push("Richard");
17 aTest.someMethod(); // Mike,Anil,Richard -- Testing
18
19 var anotherTest = new Test ("-- Testing");
20 anotherTest.variable2.push("Jurgen");
21
22 // Uses the same array from the prototype (Richard is included)
23 anotherTest.someMethod(); // Mike,Anil,Richard,Jurgen -- Testing
24
25
26 // variable2 changed on ALL instances:
27 aTest.someMethod(); // Mike,Anil,Richard,Jurgen -- Testing
28
```

Jurgen

March 20, 2013 at 11:09 am / Reply

Oh wow, that makes a lot of sense. Thank you for clearing this up, I learned a lot today!

kishore

July 2, 2014 at 9:03 am / Reply

I have small doubt. If I override someMethod function in anotherTest object, will it effect in aTest object? can I override like below code
anotherTest.prototype.someMothod = function(){
console.log("hi")
}

Tong

January 12, 2015 at 2:30 pm / Reply

anotherTest.prototype will be undefined.

Peter

July 10, 2013 at 7:35 am / Reply

This might be a stupid question, but can this be considered as a static class?

Richard Bovell (Author)

July 11, 2013 at 8:06 pm / Reply

Hi Peter,
Never a stupid question, Mate.
By "this" what specifically are you referring to?

Stefano Pongelli

August 12, 2013 at 1:43 pm / Reply

He meant that variables defined in the prototype are static, i.e. every 'instance' will share them.

Richard Bovell (Author)

August 15, 2013 at 2:10 pm /

You are correct, Stefano.

Stefano Pongelli

August 12, 2013 at 1:47 pm / Reply

A bit late to the party, let me add something: the main difference between using prototypes and declaring functions inside the constructor, is that if you do not use prototypes, each time you create a 'new' object you will also recreate all the functions that are inside the constructor.

The same does not apply if you use prototypes: it could mean a lot memory-wise!

Nikolaj Larsen

July 11, 2014 at 6:55 am / Reply

I always wondered what the difference was.

I guess another difference would be that the method declared on 'this' inside the constructor would yield true from the hasOwnProperty method, whereas the method declared on the prototype would be false

But if methods on the prototype cannot access private properties declared in the constructor of the class/function, whether to use one or the other way of declaring methods, would be a tradeof of whether you're able to hide properties or not. Said in another way, only methods that doesn't require access to private properties can be declared in the prototype. Am I completely off here?

Domen

March 20, 2013 at 10:32 am / Reply

I feel so bad that you put so much time and work doing this for free. Why not add non-intrusive ads on your website?

Richard Bovell (Author)

March 20, 2013 at 3:46 pm / Reply

This is very thoughtful of you, Domen. Thanks very much.

I am not sure I will add Ads to the site, but if I do, the Ads will be inconspicuous, like they are on this site: http://daringfireball.net/

And it is great that you are studying JS at such a young age. All the best with your career.

I plan to make some sexy T-shirts for the site :), so this will help.

Anthony

March 20, 2013 at 10:34 am / Reply

@Domen, why don't you send him a donation than? I hate ads >_<

Domen

March 20, 2013 at 11:33 am / Reply

I'm 18 years old student, who is moving away from parents in 3 months, without job nor credit card (paypal) to make a donation with money I currently have. 😛

If I get credit card anytime soon, I will certainly make a donation. 🙂

Anthony

March 20, 2013 at 11:43 am / Reply

I'm joking, sorry about that :p
Awesome that you are learning JS though!

Anthony

March 20, 2013 at 8:18 pm / Reply

Grammar fix:
"And it this need for encapsulation that we are concerned with and why we are using the Combination Constructor/Prototype Pattern."
Should be this:
And it is this need for encapsulation that we
are concerned with and why we are using the
Combination Constructor/Prototype Pattern.

Richard Bovell (Author)

March 21, 2013 at 2:10 pm / Reply

Thanks much, Anthony, I have fixed it. Not too many people report Grammar erros, so I am always happy when they are reported, so I can fix them.

DB

March 21, 2013 at 12:00 pm / Reply

I've been doing JavaScript development for a long time and this is one of the best explanations of OOP in JS I think I've ever seen. Good job!

Richard Bovell (Author)

March 21, 2013 at 2:11 pm / Reply

Thanks much, DB.
My goal is always to explain the complex topics in manner than anyone can understand, so I feel that my goal for this post has been accomplished, after reading your comment 🙂

Eric Elliott

March 21, 2013 at 1:24 pm / Reply

You're overcomplicating a few things here, and putting too much emphasis on constructors. Prototypal inheritance gets even simpler in JavaScript. See http://ericleads.com/2013/02/fluent-javascript-three-different-kinds-of-prototypal-oo/

Anthony

March 21, 2013 at 1:50 pm / Reply

I have to disagree with you, these examples are for learning and not for showing how terse JavaScript can be

Eric Elliott

April 7, 2013 at 2:01 am / Reply

It's not about terseness. It's about avoiding the coupling between the child and the parent. There is no tighter

coupling than that created by the relationship between child and parent objects, and tight coupling is exactly what you want to avoid to get flexible OO design.

I'm giving a talk on this at the O'Reilly Fluent conference titled, "Classical Inheritance is Obsolete: How to Think in Prototypal OO".

Come check it out: http://www.dpbolvw.net/click-7037282-11284265

Or watch my blog for video.

Anthony

April 7, 2013 at 6:28 pm / Reply

Eric I read your blog too, I also started reading your book on Oreilly's rough cuts. I've asked you several times for a discount and you have ignored me on each occasion to.
I find it a little rude that you are soliciting Richards readers in the comments.
JavaScript is a highly expressive language and there are a dozen ways to get to a certain end result and I appreciate different viewpoints on this. I give you big credit for your work but I also love how Richard writes and more importantly communicates with his audience, something you should probably try and do more of . I do apologize for my remark on javascript being terse, I jumped the gun before I read the post in it's entirety. They both are good examples of OO in JavaScript and I thank both of you with all sincerity

Eric Elliott

April 7, 2013 at 7:39 pm /

Hi Anthony,

I do respond to comments on my blog, but between the book writing and my full-time gig, I can't always respond the same day. I know the comment you're referring to, and I did respond. See: http://ericleads.com/javascript-applications/#comment-8900

I also communicate with readers extensively. I actively engage the community at large, including my comments on other blogs (like this one). I speak at conferences, and hang around and answer questions in person. I am an active member of the popular JavaScript communities on Google+. I engage with readers on Facebook (I maintain a page for my book with ~6k fans), I've scheduled a whole week to answer questions from readers (and potential readers) on the CodeRanch forums, and I frequently answer questions on freenode IRC in ##javascript, #node, and #jquery. I don't think I can squeeze any more engagement in, but I'll try. Thanks for the suggestion. 😉

The reason I felt the need to post a response here is because Richarc' `

claiming that his preference for inheritance is the “best”. I call these strong statements comment bait. He opened the door, so I walked in.

Some people agree with him. I don’t. Dissenting opinions are a good thing. It should encourage all of us to do more critical thinking, and hopefully question our own understandings of the language, and obtain a better understanding as a result.

I think Richard’s blog is great. I’m glad he maintains it, and I’m glad he’s open to feedback from blow-hards like me. 😉

He obviously put a lot of thought into this article, and I can’t deny that it could help people gain a better understanding of working with objects in JavaScript.

I hope both of you stay engaged, and I look forward to our next conversation. =)

Anthony

April 7, 2013 at 7:52 pm /

Cheers Eric

Eric Elliott

April 9, 2013 at 8:43 am /

Anthony, in response to your comment about not engaging enough, I've redesigned my blog with links to my social media pages, and the ability for users to register for my newsletter to get JavaScript news, my upcoming speaking events, webinars, and other opportunities to interact.

How did I do?

Richard Bovell (Author)

March 21, 2013 at 2:20 pm / Reply

Eric, you could be correct that simple Prototypal Inheritance on its own is probably "simpler" (for some developers) than the Parasitic Combination Inheritance Pattern I discuss in the article. But note that I discuss only the **best** pattern for inheritance in JavaScript.

Indeed, there are many patterns for implementing inheritance in JS, such as Factory, Parasitic, Rent A Constructor, Prototype Chaining, Combination Inheritance (the most popular of the lot), Prototypal, and others. It is up to the developer to decide which is easier, but I only discuss the best (most optimal) pattern. I don't think it is necessary for developers to know and learn **all** the patterns or the simplest (not the most optimal) one.

To learn more about the advantages and disadvantages of each pattern, read Zakas's *Professional JavaScript for Web Developers* (Chapter 6) and Stefanov's *JavaScript Patterns* (Chapters 5 and 6).

Eric Elliott

April 7, 2013 at 12:42 am / Reply

Hi Richard,

What was your criteria to decide the “best”?

I have read both books, and written about JavaScript inheritance in my own O’Reilly book, “Programming JavaScript Applications”.

Richard Bovell (Author)

April 8, 2013 at 4:55 pm / Reply

Eric,

In Professional JS for Web Developers (3rd Edition), Nicholas C. Zakas was explicit when he described the best (“most most optimal”) two patterns. And he painstakingly explained how those two patterns improved on all the other patterns, comparing the advantages and disadvantages of each.

I have not personally researched all the **many patterns** and test and compare each, so I am not qualified to debate this topic at length.

Eric Elliott

April 9, 2013 at 8:47 am /

Yeah, he did discuss a lot of patterns, but one thing they all had in common was that they were mostly based on using prototypal inheritance to mimic the techniques used in classical OO — specifically, the establishment of the parent/child links.

Those links though are completely unnecessary in prototypal langu

capable of dynamic object extension. Take a look at my stampit examples and see if you can figure out why I like it better.

Richard Bovell (Author)

March 25, 2013 at 2:06 am / Reply

Eric,
I am looking forward to your “Programming JavaScript Applications” books. The outline looks good.

Also, your post, “Switch … Case Considered Harmful,” is an informative post, I have never seen objects (method lookup) being used in place of switch statements:
Switch … Case Considered Harmful

Anthony

March 27, 2013 at 4:02 pm / Reply

Crockford doesn’t like using switch in JavaScript as well, I can’t find the post where he talks about it though.

Eric Elliott

April 7, 2013 at 8:39 pm / Reply

I believe Crockford addressed his concerns about switch statement fallthrough in “The Good Parts”. However, he still uses switch statements in his code (to my knowledge), and simply avoids using the fallthrough feature.

Interestingly, a friend of mine informed me that fallthrough will throw an error in C# compil

Apparently we're not the only ones who think it's problematic. =)

Anthony

April 7, 2013 at 8:44 pm /

They work in AS3, but even though they look sexy, I avoid them in JavaScript because of what Crockford said.
Do you know why they are so problematic Eric?
do you know when that video is going to up of the Fluent conference too?

Eric Elliott

April 7, 2013 at 8:53 pm /

Fallthrough is problematic primarily because it's really easy to forget the break, and because you might leave break off intentionally when you employ fallthrough, a forgotten break is much harder to catch than say, a forgotten bracket.

Crockford argues that you should never use the fallthrough feature, so you know that any forgotten break is an error.

I argue that switch statements in general get abused and messy as cases get added, and eventually

into a giant spaghetti bowl, whereas method lookup tends to do a better job of ensuring that all the cases are related. At the same time, method lookup allows for more flexibility and better extensibility.

Given that they're also faster than switch statements, and have no fallthrough feature to worry about, I don't see a compelling argument to ever use switch in JavaScript. I haven't written a switch statement for production code for several years.

Anthony

April 7, 2013 at 9:05 pm /

Slow, I don't like that word lol
every little bit helps, because it's the little things that add up and clobber you when your app grows. Yea I don't use them neither

Eric Elliott

April 7, 2013 at 9:19 pm /

It's really funny that you mentioned slow in this context. I just finished writing a blog post about knowing what to optimize. http://ericleads.com/2013/04/youre-optimizing-the-wrong-things/

Richard Bovell (Author)

April 8, 2013 at 5:05 pm /

Great article, Eric. I just tweeted about it.

Eric Elliott

April 9, 2013 at 8:39 am /

Thanks, Richard.

Eric Elliott

April 7, 2013 at 2:02 am / Reply

Hi Richard,

I'm participating in the O'Reilly Blogger Review program if you'd like to get an advance copy for review. http://oreilly.com/bloggers/

Richard Bovell (Author)

April 8, 2013 at 4:59 pm / Reply

Ok, thanks for the invite, I will give it a go. I actually need a better book than "JavaScript Patterns" to recommend for my Learn Advanced JavaScript post. I am not completely happy with that book, and none of the other (Adv. books) out there is much better.

There is no single great advanced JavaScript book that I have seen. So if your book is great (or even better than just the JS Patterns book), I will replace the JS Patterns book with it on the Learn Adv JavaScript post.

Eric Elliott

May 2, 2013 at 5:06 am /

Richard, if you haven't had the chance to read my book yet, I'd love to hook you up with a copy for review. Drop me a note. http://ericleads.com/contact-form-wts/

Richard Bovell (Author)

May 2, 2013 at 11:06 am /

Okay, I will. I have been quite busy lately.
I have 3 books to review in the next 2 weeks, and I will review yours at the same time.

antonio

March 21, 2013 at 2:35 pm / Reply

First off, thanks for creating such a informative and well designed website. It's really slick. I have been trying to learn for JS for a while, but only recently have been making progress…(Followed your post on "how to learn js" and just purchased Professional JS for Web Developers).
Question: I have read in many arenas creating methods via the prototype is the way to go. The rational is the subsequent new Object will inherit those methods only once, (and there's a memory issue happening)rather than if one had placed said method in a constructor. So that can one use it with properties too?
If so why wouldn't we always create our objects via prototypes? Seems like an extra step?

Richard Bovell (Author)

March 27, 2013 at 3:47 pm / Reply

Hi Antonio, I am not sure I understand your question. Could you rephrase your question? I would be happy to answer you once I understand precisely what you mean.

Thanks.

antonio

March 27, 2013 at 9:16 pm / Reply

Sorry, I wrote that at work!
From what I have gleaned from other resources, you use the Constructor pattern to make objects. But I have also read, when you're giving that Constructor a method, you don't want to make it in the Constructor, you use it's prototype (which every function has inherently) instead. The reason being, if you create the method in the original Constructor, when you create a new object from that Constructor that method would be copied every time, causing memory problems.
So I guess my question is why wouldn't you create the object using the prototype pattern in the first place?

Richard Bovell (Author)

March 27, 2013 at 11:24 pm / Reply

That is a very good question, Antonio.

First, in the examples (the two patterns we use) in the article above, note that we **are** using the prototype to define the methods and properties. But we are using more than just the prototype,

because the Prototype Pattern alone has some disadvantages.

The main disadvantage with the Prototype Pattern is that all the properties are inherited and shared by all the instances, and when you change a prototype property on one, it is reflected on all of them. This is particularly true for reference types. Look at the discussion Jurgen and I had above. I touched on this. He had a similar question.

Also, neither of the two patterns I use above use the full Constructor Pattern. They only use the Constructor to define instance properties. There is no memory problem with the Constructor properties. Only Constructor methods can cause memory issues.

But note that I only have one method defined in the Constructor (this.getQuizDate), and it is not necessary. I only added it there as an example of a private method. All it does it show the date the quiz was created. We can easily add it on the prototype.

Your point is noted though, because you are correct that **methods** defined in the Constructor will be recreated for each instance, and thus use memory. So if you have a lot of instances and a lot of Constructor methods, then the memory issue could be a problem. We don't have this problem, since we only have the one *unnecessary* (this.getQuizDate) method defined in the Constructor.

Anthony

March 21, 2013 at 7:46 pm / Reply

Finally got a chance to go through the code more thoroughly, and caught another typo Richard. in the Question constructor you have an extra var keyword in front of the QUIZ_CREATED_DATE variable/constant.

Thanks again for tour of OO inheritance in JavaScript, are you going to a take on the prototypical qualities of the language ?

Anthony

March 21, 2013 at 7:48 pm / Reply

I mean another take on how Prototypical works in JavaScript o_0

Richard Bovell (Author)

March 25, 2013 at 1:43 am / Reply

Both of the techniques I cover in this post use Javascript prototype, and I have a through post on just JavaScript Prototype, “JavaScript Prototype in Plain Language”: http://javascriptissexy.com/javascript-prototype-in-plain-detailed-language/

Anthony

March 27, 2013 at 3:59 pm / Reply

I realized that after I wrote the post lol, too much code and not enough sleep does strange things to ones mind lol
what about the typo, I do recall that was an issue?

Richard Bovell (Author)

March 27, 2013 at 4:29 pm /

Get enough sleep, Mate. I know a lot of programmers go with little sleep, but sleep is very important.

I missed your first post about the double var. That was a very good catch, especially because it was a subtle mistake. I just fixed it.

Bhawani Singh

March 22, 2013 at 7:56 am / Reply

Nice post!!!!!!!! keep it up

Sung Am YANG

March 22, 2013 at 4:42 pm / Reply

Thanks. I found that your blog is very useful. How can I keep informed of your posts instead of visiting here manually?

Richard Bovell (Author)

March 25, 2013 at 1:45 am / Reply

Many users have inquired about a mailing list, but I don't want to send out emails. You can follow me on Twitter, where I announce when I publish a new article. Or you can check the box "Notify me of new posts via email" that you see when you are adding a comment.

Tyrone Michael Avnit

March 23, 2013 at 9:04 am / Reply

One of the more informative articles I have read lately. Thanks for explaining it with such ease. Will definitely be subscribing the the blog.

Richard Bovell (Author)

March 25, 2013 at 2:19 am / Reply

I am very happy to hear that the post is informative. Thanks for the kind words.

Thanks also to @Bhawani above.

Abhinav Arora

March 24, 2013 at 4:02 pm / Reply

Hi Richard…amazing work. this is surely the best thing out there on JS. I have been learning javascript like you said. Could you please suggest me some open source projects where I can practice and hone the newly aquired skills.

Appreciate the effort. Hats Off to you man!!

Richard Bovell (Author)

March 25, 2013 at 1:58 am / Reply

Thanks much, Abhinav.
I don't which specific project would be ideal for you, but I recommend you look for "javascript" projects on GitHub, of which there are quite a lot. And most them need contributors.

Henrique A. Silvério

March 31, 2013 at 12:37 pm / Reply

Hey Richard, thanks a lot for these hight quality informations. I'm curious when you say "inheritance and encapsulation" only applicable on OOP in JavaScript. After I read a article about polymorphism in OOP JS here:

http://www.codeproject.com/Articles/315169/Polymorphism-in-JavaScript. What you think about it? Thanks!

Richard Bovell (Author)

April 1, 2013 at 8:58 pm / Reply

Many OOP purists like to add OOP principles to JavaScript that are common in languages OOP languages like Java. But it is unnecessary to do this in JavaScript, and the result is usually complex code that is not always productive.

Suffice to say, these OOP principles like polymorphism and Interfaces, amongst others, are not necessary in JS and they are't worth the time and effort for JS developers.

Nick

April 26, 2013 at 10:32 pm / Reply

Great article and I expanded my js knowledge even more. I have been writing some semi-complicated applications and it always seems to string along so I decided to read up some and this article will definitely come in handy. Although I do have one question. When making the MultpleChoiceQuestion you did this inside the function:

Question.call(this, theQuestion, theChoices, theCorrectAnswer);

Now when I was playing with this myself I found that I could just do:

this.constructor.apply(this,arguments);

I was wondering if there were any drawbacks to this or any reason why you didn't do this. In my head I am thinking it is better because it will use whatever constructor you tell it to use (so if you were to change constructor wouldn't have to worry about updating the calls everywhere) and it will also allow you to add additional arguments without having to update each of the inherited classes. I

am not sure if there is something I am missing but I figured it was worth seeing if you had any input on this.

Richard Bovell (Author)

April 28, 2013 at 7:51 pm / Reply

Nick,

The style you have written for handling the arguments is totally fine. It is mostly used for variable-arity functions (functions that can take any number of arguments). And you see it quite often in JS libraries.

So, go for it, since you like that style.

Tyler

May 3, 2013 at 12:23 am / Reply

I know you are busy, but I miss reading your blog. Looking forward to your next post!

Richard Bovell (Author)

May 3, 2013 at 12:43 am / Reply

Tyler, I am working on my next blog post right now. I am hopeful I will complete it tomorrow. I think I will.

Thanks for the encouragement.

Gopala Krishna Bala

May 14, 2013 at 7:34 am / Reply

I am a web developer, wants to receive the updates

Richard Bovell (Author)

May 14, 2013 at 11:20 pm / Reply

Thanks for your interest, Gopala. I will add a newsletter to the blog very soon.

Alex Hurtt

May 22, 2013 at 8:52 am / Reply

I am curious as to why you choose the approach of overwriting the implicit prototype of a function/object with your own anonymous object like so:

```
function F() {}
F.prototype = {
constructor:F,
someFunction:function(){}
}
```

as opposed to just this:

```
function F() {}
F.prototype.someFunction = function() {}
```

In other words, what is the advantage or at least the reason behind the choice of overwriting/redefining the implicit prototype object with one of your own creation and having to explicitly delare the constructor property?

Richard Bovell (Author)

May 28, 2013 at 1:39 am / Reply

Great question, Alex.
First, it is important that I clarify one note in your comment: the Prototypal Inheritance function that you inquired about, which is shown in code below, was created by the venerable JavaScript master Douglas Crockford. I did not create it.

Here is the code again, for clarity:

```
1  Object.create = function (o) {
2          function F() {
3          }
4
5          F.prototype = o;
6          return new F();
7      };
8
```

I think the reason Crockford overwrites the entire prototype is to implement a full copy of the “o” object, and thus the new F () object that is returned inherits completely from the “o” object (the F object is a shadow copy of the “o” object).

Simply put, you pass in an object to the function when you want to make a full copy of it (or to inherit fully from it). This is the reason for the full prototype overwrite, I presume.

Eric Elliott

May 28, 2013 at 6:55 am / Reply

That function simply creates a new object with whatever object you pass in as its prototype.

F() is a constructor function that gets called with `new` to instantiate a new object that has a reference to whatever object you pass in as “o”.

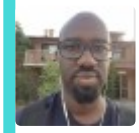
In other words, to inherit from some prototype, “foo”, you would do:
var bar = Object.create(foo);

“foo” is “o” inside the function.

Rhode Island Web Design

June 9, 2013 at 5:46 am / Reply

JavaScript is an excellent language to write object oriented web applications. It can support OOP because it supports inheritance through prototyping



Elie GAKUBA

July 7, 2013 at 10:50 pm / Reply

Thank you Richard. It is a good practice.

Gopal

July 8, 2013 at 12:33 am / Reply

Hi Richard

I have been following your posts since last week. Really these are superb. i have learned a lot from your posts. Now i get a confidence that i can implement OOP concepts in my work. Please update me about your new posts

keep it up good work.

Thanks

Richard Bovell (Author)

July 11, 2013 at 9:10 pm / Reply

Glad to hear, Gopal.

And thanks, Elie.

Ashish

July 15, 2013 at 6:02 am / Reply

Awesome… thank you so much!!

Ari

July 15, 2013 at 8:14 am / Reply

"// Override the createQuestion method it inherited"

did you mean to say...
// Override the displayQuestion method it inherited

BTW, thanks for the awesome explanations backed by appropriate code!!!

Richard Bovell (Author)

July 17, 2013 at 3:19 pm / Reply

Thanks much, Ari. You are correct, I just fixed it. Good catch.

Terral Lewis

July 24, 2013 at 9:29 am / Reply

Is the Object.create method used in conjunction with the inheritPrototype function or can one or the other be used? Seems to me they virtually do the same thing and if not, could you explain the differences? Thx!

Richard Bovell (Author)

July 31, 2013 at 2:14 am / Reply

Very good question, Terral.

The Object.create method basically copies the methods and properties from one object to another, and it returns a new object that inherits from the object you passed into it.

On the other hand, the inheritPrototype function uses the Object.create method to do the copying of methods and properties

and then it also reset the Constructor and the prototype because the Constructor gets overwritten during the Object.create step.

So, the direct answer to your question:
you can use **only** the inheritPrototype function, since it makes us of Object.create to copy the methods and properties.

Ryan

July 30, 2013 at 6:36 pm / Reply

Thanks for all this! I have a question:

I've been trying to get the code:

```
allQuestions.forEach(function(eachQuestion) {
eachQuestion.displayQuestion();
});
```

to display all the questions in HTML. If I console.log(questionToDisplay); in the original Question.prototype, everything shows up, but trying a method to get it to display in HTML always results in undefined.

Any suggestions?

I really appreciate it. I'm a designer, and your track is the first to really get Javascript to click.

Richard Bovell (Author)

July 31, 2013 at 5:17 am / Reply

First, thanks very much for helping me fix the issue with the blog blocking comments, Ryan. I really appreciate it.

Could you post all of your code here or on JSBin so I can see what is causing the error?

Ryan

July 31, 2013 at 7:01 pm / Reply

Sure thing:

http://jsfiddle.net/mickhavoc/dDE92/27/

Richard Bovell (Author)

August 1, 2013 at 12:27 am / Reply

There are 2 simple things missing in your code:
1. In the displayQuestion method, you were not actually returning anything. You commented out the console.log statement, but you forgot to add this:

```
1
2  // You have to return the questionToDisplay variab
3  return questionToDisplay;
4
```

2. And when you call the displayQuestion method, remember you have to execute it as a function with the parentheses like this:

```
1
2  allQuestions[0].displayQuestion();
3
```

You were calling it like a property like this:

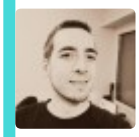
```
1
2  allQuestions[0].displayQuestion;
3
```

So those two small changes will get your code working.

One other note: as you will see in your JSFiddle example, the questions are being displayed fine, but they are appended to the HTML document as **text**. This makes sense because of this line:

```
1
2 document.createTextNode(allQuestions[0].display(
3
```

You have to modify the displayQuestion method to return the question without the HTML div stuff. You can add the HTML div stuff separately with createElement.



Stefano Pongelli

August 12, 2013 at 1:50 pm / Reply

Very nice article indeed!

abhishek

August 23, 2013 at 7:38 am / Reply

hey, I got the CC/PP, the first part and thought like got the Crockford's Object.create too the way you explained. But then, I lost it quickly in inheritProperty function.

var copyOfParent = Object.create(parentObject.prototype);

//Then we set the constructor of this new object to point to the childObject.
//This step is necessary because the preceding step overwrote the childObject constructor when it overwrote the childObject prototype (during the Object.create() process)

why you said that preceding stpe overrworte the childObject contructor when it overwrote the childObject prototype (during the Object.create() process)

basically the last comment of yours, as I see preceding step is doing nothing with the childObject at all.

And also might be I don't get it all. For me
you should be passing parent Object in Object.create here instead of parentObject.prototype
var copyOfParent = Object.create(parentObject.prototype);

lets say in the simplest of scenario, when parentObject is created from literal or new Object() prototype would be Object right? and we don't get any properties of parentObject at all only Object properties we will get which is nothing.

Its like in the example just above you gave if you have passed cars.prototype would you have got anything (any property I mean) in toyota?

Richard Bovell (Author)

August 23, 2013 at 12:28 pm / Reply

> *why you said that preceding stpe overrworte the childObject contructor when it overwrote the childObject prototype (during the Object.create() process)*

Ordinarily, the constructor property of every function points to the constructor of the function. For example:

```
1 function Fruit () {}
2 var newFruit = new Fruit ();
3 console.log (newFruit.constructor) // Fruit ()
4
```

But when we overwrite the prototype like we did in the example code in the article, this happens:

```
function Fruit () {}
Fruit.prototype = {
myName: function () {
console.log("My Name function");
}
}

// Here, you see that the constructor property no longer points to Fruit ()
// it has the value of the Object () constructor instead
var anotherFruit = new Fruit ();
console.log (anotherFruit.constructor) // Object()

```

So, if we want the constructor property to point to the correct constructor, we have to reset it. Thus: this line:

```
constructor: User

```

> *And also might be I don't get it all. For me*
> *you should be passing parent Object in Object.create*
> *here instead of parentObject.prototype*
> *var copyOfParent =*
> *Object.create(parentObject.prototype);*

The reason we pass the parentObject.prototype to the Object.create method is because we want the copyOfParent prototype to inherit from the parentObject prototype. The last line is this: childObject.prototype = copyOfParent;

The prototype properties will not be inherited in the example code you provide.

Note that you can inherit properties with this:

```
1  var someObj = Object.create (anotherObj);
2
```

Or you can inherit the prototype properties like this:

```
1  var someObj.prototype = Object.create (anotherObj.prototype);
2
```

> *lets say in the simplest of scenario, when parentObject is created from literal or new Object() prototype would be Object right? and we don't get any properties of parentObject at all only Object properties we will get which is nothing.*

Yes, you are correct that the parentObject's prototype will be the Object () constructor. But parentObject will get some properties because there are many properties and methods that are inherited from the Object prototype.

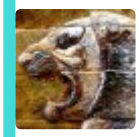
These inherited properties and methods that objects inherit from the Object () are constructor, hasOwnProperty (), isPrototypeOf (), propertyIsEnumerable (), toLocaleString (), toString (), and valueOf (). ECMAScript 5 also adds 4 accessor methods to Object.prototype.

> *Its like in the example just above you gave if you have passed cars.prototype would you have got anything (any property I mean) in toyota?*

You should understand this now, after you have read my answer to the first part of your question.

To understand Object.create, know that if we pass in an object, we are inheriting the properties of that object. If we pass in an object prototype, we are inheriting the object prototype like this:

```
1
2  copyOfParent.prototype = Object.create(parentObject.prototype);
3
```



Dave Rothfarb

September 28, 2013 at 1:26 pm / Reply

Thank you so much for this fantastically helpful guide! I'm a JavaScript newbie and coming from lower level OOP languages, I need all the help I can get. This is a great site!

-Dave

Foysal Mamun

September 30, 2013 at 1:50 am / Reply

Thank you very much to describe easy way.

David

October 9, 2013 at 11:22 pm / Reply

Hi! ,i have a question if i want call a function of the same class from other function also of that class, What do I do?.

for example

function Myclass(){}

```
MyClass.prototype = {
constructor:MyClass,
func1:function(){ alert("hello world"); },
func2:function(){ //call func1 ??? how???? }

}
```

thanks! , very nice post!

Richard Of Stanley (Author)

October 11, 2013 at 2:08 pm / Reply

Here is the answer (you have to use the “this” keyword):

```
1
2  function MyClass(){}
3
4  MyClass.prototype = {
5  constructor:MyClass,
6  func1:function(){
7    alert("hello world");
8      },
9  func2:function(){
10   this.func1();
11  }
12 };
13
14 var aClass = new MyClass();
15 aClass.func2(); //should show alert  window, since it calls func1
16
```

david

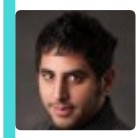
October 13, 2013 at 8:07 pm / Reply

thanks!

Viva Victor

October 12, 2013 at 8:32 am / Reply

I enjoyed going through your article, i'ts really good kip it up. A lot of young programmers (like me) are outside looking for someone like you to make us understand more. Thanks!

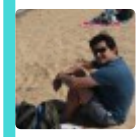


Mahdi Pedram

October 16, 2013 at 7:18 pm / Reply

hey I would swap those two lines, the contructor is getting overridden by the object.prototype. so you always need to put set the constructor at the end.

```
function inheritPrototype(childObject, parentObject) {
var copyOfParent = Object.create(parentObject.prototype);
childObject.prototype = copyOfParent;
copyOfParent.constructor = childObject;
}
```



Soichi Hayashi

October 24, 2013 at 9:58 am / Reply

Thank you for the great article!

Could I (or should I?) make all inherited functions defined inside the constructor as privileged functions, instead of defining them outside the constructor function using .prototype property? Is there a fundamental difference in the way privileged functions are inherited compared to prototype methods?

Richard Of Stanley (Author)

October 29, 2013 at 2:41 pm / Reply

That is a very good question, Soichi.

Jurgen asked the same question and I posted a detailed answer. It is just above this, you have to scroll up a bit. Here is the link directly to the question and my answer:
http://javascriptissexy.com/oop-in-javascript-what-you-need-to-know/#li-comment-4277

William

October 26, 2013 at 11:17 pm / Reply

Hi Richard, Thanks for writing these articles. You're doing something online that no one else is doing.

I've been thinking about the different ways to do inheritance. I think I understand the constructor/prototype pattern, and PCI.

My question is, Have you seen Resig's script for inheritance? He made some helper functions much like the ones in the PCI pattern.

http://ejohn.org/blog/simple-javascript-inheritance/

What're your thoughts on his pattern?

Richard Of Stanley (Author)

October 29, 2013 at 3:44 pm / Reply

There are so many inheritance patterns in JavaScript that to study and compare and contrast them alone could take an entire book.

The pattern that John described in his article is (or perhaps was) popular, probably because John made it famous, given his well-regarded status in the JS community. I still see it from time to time in some some JS libraries

One day I will compare and contrast some of the popular patterns, hopefully sooner rather than later. And only then can give a more through comparison of that pattern and the one in this article.

For now, though, I looked at the pattern and it is actually very similar to the pattern I described above, with some added complexities.

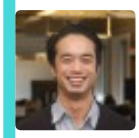
William

October 29, 2013 at 4:21 pm / Reply

Thanks for reply. Yea there's a lot of Inheritance patterns out there.

I seem to like Resig's pattern since his _super method allows you to tweak the subclass' namesake method in a concise manner. In the constructor-prototype pattern, you sorta have to rewrite the whole thing out, and then tweak it.

Alex Rauschmayer also wrote a similar script which was meant to improve on Resig's (https://github.com/rauschma/class-js). This one uses Object.create.



Sean

November 8, 2013 at 11:17 pm / Reply

Richard,

I just wanted to say a BIG THANK YOU for all of your articles, including this one. I just spent the last few hours digesting this.

I'm currently working thru the "Learn Javascript Properly" Course on reddit, and I must commend you for the GREAT WORK you are doing here. Your blog posts simplify what can is otherwise dense material. I've tried to keep up with the course materials (which uses Zakas' Professional Javascript for Web Develop

and must say that your articles are much more accessible than the text book for novices such as myself.

I've already read more than a handful of your articles on this site and every single one has been awesome. You have a remarkable talent for simplifying concepts. Hopefully I will continue to find articles made by you that match up/supplement/are in conjunction with the syllabus of reading materials in the Learn JS Properly course, because your articles have made all the difference for me – and that is one of the biggest compliments I can give someone.

So again thank you, and I'm sure I'll continue to frequent this site (I have it bookmarked multiple times).

PS: you should consider writing your own book! Seriously! Or at least start w/ more and more blog posts/articles that follow a roadmap/blueprint for a beginner to learn JS. I know this exists already: http://javascriptissexy.com/how-to-learn-javascript-properly/

But, maybe instead of using those textbooks, you write the material for beginners to read and learn from (cuz I find your teaching style much more facilitative to learning, and judging from the 100+ comments on each one of your posts, others do too).

AGAIN, sorry for the long post, but I just wanted to express my gratitude cuz I can tell each and every one of these articles takes a lot of effort, time and intellect to write. Least I can do is take 15 minutes to thank you.

THANKS!

Richard Of Stanley (Author)

November 10, 2013 at 2:40 am / Reply

Sean, You have made my day. Thanks very much for the wonderful compliment and advice.

I do plan to write a book indeed, but I am extremely busy creating a startup right now, which I plan to reveal a demo video of very so

fact, it will provide a lot of direct learning material for web dev technologies. So stay tuned.

I agree that some of the textbooks are a bit terse and sometimes the blog articles are more friendly. I will see how best I can address this in my startup.

Thanks again and take care.

William

November 10, 2013 at 10:30 am / Reply

Sounds exciting.. really looking forward to seeing what you're making. Are you going to be sending out the launch email through this blog's listserv? Please do!

Thanks again all of the articles your write here, Richard. Peace!

Richard Of Stanley (Author)

November 22, 2013 at 6:46 am / Reply

Yes, we will definitely send out the launch email via the mail chimp mail list.

srinath

November 23, 2013 at 1:30 am / Reply

Hi you have done a good job.Nice Artical.
I have One doubt
How to display all quiz questions in a div without console.log?

Srinath

gavin

November 23, 2013 at 3:12 am / Reply

Great article thanks. Just one thing I notice is choiceCounter is not declared so it will be set in the global scope. Is that intended or should you place that inside Question as extra instance property and accessed using this.choiceCounter in Question.prototype.displayQuestion

John

November 26, 2013 at 5:14 am / Reply

function inheritPrototype(childObject, parentObject) {
// As discussed above, we use the Crockford's method to copy the properties and methods from the parentObject onto the childObject
// So the copyOfParent object now has everything the parentObject has
var copyOfParent = Object.create(parentObject.prototype);

//Then we set the constructor of this new object to point to the childObject.
//This step is necessary because the preceding step overwrote the childObject constructor when it overwrote the childObject prototype (during the Object.create() process)
copyOfParent.constructor = childObject;

// Then we set the childObject prototype to copyOfParent, so that the childObject can in turn inherit everything from copyOfParent (from parentObject)
childObject.prototype = copyOfParent;
}

I have been asking around at stack overflow and am still not getting a satisfactory answer. Please help.

When you say "//This step is necessary because the preceding step overwrote the childObject constructor when it overwrote the childObject prototype (during the Object.create() process)"
BUT how can the previous step had affected the childObject at all. All the

previous step did was create a copy of parentObject. Nothing to do with the childObject. var copyOfParent = Object.create(parentObject.prototype);

Richard Of Stanley (Author)

November 26, 2013 at 8:03 am / Reply

Good urgent question, John 🙂

I was incorrect when I said this:

> *This step is necessary because the preceding step overwrote the childObject constructor when it overwrote the childObject prototype (during the Object.create() process)*

Therefore, you are correct when you said this:

> *BUT how can the previous step had affected the childObject at all. All the previous step did was create a copy of parentObject. Nothing to do with the childObject. var copyOfParent = Object.create(parentObject.prototype);*

I should have said this (which I just updated in the article, with credit to you of course):

```
1
2  // Then we set the constructor of this new object to point to the childObje
3     copyOfParent.constructor = childObject;
4  // Why is this necessary? See the explanation immediately following this c
5
```

We explicitly set the *copyOfParent.constructor* property to point to the childObject constructor because in the preceding step, var copyOfParent = Object.create(parentObject.prototype), this is what we actually did:

```
1
2  // We made a new object and overwrote its prototype with the parentObje
3  function F() {
4        }
5  F.prototype = parentObject.prototype;
6  // Then it was this new F object we assigned to copyOfParent.
7  // All of this was done inside the Object.create () method.
8
```

So, this new F object, which we assigned to copyOfParent, doesn't have a constructor property anymore because we overwrote its entire prototype. Whenever you overwrite an object's prototype (object.prototype = someVal), you also overwrite the object's constructor property.

To make sure we have the correct value for copyOfParent constructor, we set it manually with this:
copyOfParent.constructor = childObject;

gavin

November 26, 2013 at 8:21 am / Reply

I think it helps to understand it better if the last step is switched with the second step. ie
function inheritPrototype(childObject, parentObject) {
var copyOfParent = Object.create(parentObject.prototype);
childObject.prototype = copyOfParent;
copyOfParent.constructor = childObject;
}

Because the original childObject.prototype has constructor set to 'childObject' but was overridden to be 'parentObject' from the assignment in the second line. This is because the 'copyOfParent' is a parentObject.prototype clone which has parentObject.prototype.constructor=parentObject by default.
Hence the last step is to set the constructor back to the childObject

John

November 26, 2013 at 10:28 am / Reply

makes perfect sense now. thanks.

btw I sent u this on twitter.

"Also, advise everywhere is that rather than finding classical child-based inheritance in JS, one should be looking at interfaces & mixins. So can u pretty pls write an article on js interfaces and mixins?"

Kiran

December 5, 2013 at 7:05 am / Reply

Hello Richard,

Thank you very much for your neat write up 🙂
Finally, I have learned that there are more to learn in Javascript world 🙂
I have a few questions, please bear with me if it sounds silly.

```
function A() {
}
a = new A();
function B() {
}
B.prototype = a;
```

1) "a" and "A" are both treated as objects in Javascript right..
"A" points to the definition of "A" itself

"a" has a reference to a memory space having a copy of template represented by "A".
Is this correct for Javascript as well? This is my basic understanding of Class/object relation?

2) Can you explain why the following happens?
B.prototype = A //does not work – can't access B's properties via "a"
B.prototype = a //works

Thanks a lot in advance.

Regards,
Kiran

Rusty

December 5, 2013 at 8:40 pm / Reply

Hi Richard,
Its nice article. I have searching through lot of articles but this one made my doubts clear. Also thanks to comments providers for clearing some point.
Just one thing added to article could make it more sexy as your site says Javascript is SEXY.
Graphical Presentation… would like to see more images and diagrammatic representation that will make things to understand better and quick to remember.

Thanks for your efforts on availability of these information and knowledge sharing.

Praga

December 15, 2013 at 2:39 am / Reply

Hi Richard,

I have a doubt.

I read that private variables [this] cannot be used inside prototype functions.

But you have used [this.quizscores] inside prototype function under Encapsulation topic.

Could you please explain how it works fine?

Anthony Howarth

January 16, 2014 at 8:12 am / Reply

Great JavaScript articles! At first, I found the inheritPrototype(ChildObject, ParentObject); a little confusing, because of the intermediate copyOfParent object.

I rewrote the function to get rid of copyOfParent. The function now seems a lot clearer as to what it is doing, and it's a little shorter too:

```
function inheritPrototype(ChildObject, ParentObject) {
ChildObject.prototype = Object.create(ParentObject.prototype);
ChildObject.prototype.constructor = ChildObject;
}
```

Just my two cents …

Shesh

May 18, 2015 at 5:50 pm / Reply

Agreed Anthony, that temp object is unnecessary (it appears like an intermediate object we are using in swap examples, but in reality it is not!) , I hope Richard makes the correction.

Neven

May 6, 2014 at 9:04 pm / Reply

Hello Richard,
thank you for your post, i am new to OOP JavaScript and this post is very helpful.
I have some question:
1. Why do we have to set constructor back to childObject ? can you give some issues if we do not set constructor back ?
2. In Question constructor and inheritance, why do you have to implementing inheritance by seperating the code in each line Question.prototype ? Why don't you use object literal to override like User constructor for short ?

Bo

June 2, 2014 at 1:54 am / Reply

Great as always – I come back to this post from time to time for a quick refresh – for while I do own (and have read in most cases) the books referenced I find your talent in communicating these subjects to be just perfect. Quick question – building large AngularJs app, utilizing one of the many BaaS providers (Firebase in this case) and building out objects to include the models (i.e user, project, asset etc.) while there is limited commonality across these models some does exist – so question(s): Is there a 'best practice' for approaching this? meaning would it be best to have the highest object with the 3 common properties and inherit fromt there? or, better to simply remain within each functional model, and if possible share inheritance down the line there? Also, including each field in the args (i.e. fullname, email, lastname etc.) as this limits the ability to inherit from that object is there a better way? thanks for your time – and again, thank you for providing this fantastically valuable guidance

Federico

June 23, 2014 at 8:10 pm / Reply

Hello, I read the article and it's very nice.

Instead of using this function:

```
function inheritPrototype(childObject, parentObject) {
var copyOfParent = Object.create(parentObject.prototype);
```

```
copyOfParent.constructor = childObject;
childObject.prototype = copyOfParent;
}
```

couldn't just use something like this?

```
function myInheritPrototype(childObject, parentObject) {
childObject.prototype = {
constructor: childObject,
__proto__: parentObject.prototype
};
}
```

Is there any drawback or issue with this approach? It is just I don't see why we need to create a new object with all those invocations. Basically here I just inherited all father's prototype by assigning it to __proto__ inside child's prototype, right?

Great work Richard!!!
Regards from Argentina

Thomas

June 24, 2014 at 6:17 pm / Reply

Could you explain the reasoning behind using the following:

```
function inheritPrototype(childObject, parentObject) {
var copyOfParent = Object.create(parentObject.prototype);
copyOfParent.constructor = childObject;
childObject.prototype = copyOfParent;
}
```

instead of below:

```
function inheritPrototype(childObject, parentObject) {
childObject.prototype = Object.create(parentObject.prototype);
```

childObject.prototype.constructor = childObject;
}

Looks like it saves a line of code, is it performance related?

Thanks in advance.

Adel

August 11, 2014 at 7:43 am / Reply

Richard, you are the best! Thank you so much, I like you man
I'm 21 years old and I've been studying this programming for less than 4 months, so I learned css and html and a lot of applications of JavaScript,and when I bumped into oop in JS I found hard to deal with, however, thanks God I found you and many thanks to you because you teached me oop in JS.

Bipul

October 10, 2014 at 12:42 pm / Reply

Hi Richard,

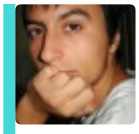
Its really the best explanation one can find on the planet.
I had a little confusion on how can we replicate function overloading in Javascript. Is it possible to have 2 functions with the same name but different parameters , inside a javascript constructor.

Bipul

October 10, 2014 at 11:52 pm / Reply

Hi Richard,

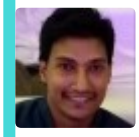
Really these were the best explanation one can find on this planet. And you are the best person who can clarify any doubt in javascript.
I had a doubt about how can we achieve function overloading in javascript?



Humberto

October 23, 2014 at 5:41 am / Reply

Thanks for this article. It's a great way to understand OOP in JS.



Venkat

October 25, 2014 at 3:06 am / Reply

In my post JavaScript Prototype, I explained that every function has a constructor property, and this property points to the constructor of the function.

I think to be more accurate:

Every function has a prototype property and the prototype will have a constructor property, and this constructor property points to the constructor of the function. is it correct?

Keith Roberts

October 26, 2014 at 8:56 am / Reply

Firstly, thank you for taking the time to write this tutorial it's very helpful to see an example of how OOP can be used in creating applications.

I'm trying to clearly understand the code, and wondered why when adding the methods to the Question.prototype you didn't use an object literal like the User.prototype. But perhaps more importantly – is there a reason why the constructor is not 'reassigned' to the Question.prototype? e.g Question.prototype.constructor = Question;

anilmcmt

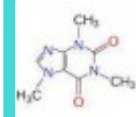
October 28, 2014 at 1:56 am / Reply

nice tutorial

Gregory Degruy

November 17, 2014 at 1:05 pm / Reply

Great overview of using OOP in js, the exercise at the end was the icing on the cake :D.



mozillanerd

November 23, 2014 at 3:54 pm / Reply

The operator ‘instanceof’ is misspelled; must be lowercase.

DeVontae Moore

December 8, 2014 at 9:05 pm / Reply

could someone please explain this:
Question.prototype.displayQuestion = function () {
var questionToDisplay = “” + this.question + “”;
choiceCounter = 0;

this.choices.forEach(function (eachChoice) {
questionToDisplay += ” + eachChoice + ”;
choiceCounter++;
});
questionToDisplay += “”;

console.log (questionToDisplay);

Thanks for your time in advance to whomever does answer.

Nhuy Van

December 16, 2014 at 4:14 am / Reply

Hi Richchard, Thank you so very much for taking the time writing these amazing posts. I just have one question though inside the inherentPrototype() functi~~

```
function inheritPrototype(childObject, parentObject)
{

var copyOfParent = Object.create(parentObject.prototype); // Why did we just pass in “parentObject” instead of its prototype, because I think you can inherit from “parentObject” that way right?

copyOfParent.constructor = childObject;

childObject.prototype = copyOfParent;
}
```

dragonslayer555

December 27, 2014 at 6:55 pm / Reply

First of, I’d like to say thank you for all of these articles! I have learned a ton since stumbling onto your blog.
You have a small typo in one of the examples.

```
function Employee () {}
...
Employee.prototype.fullName = function () {
console.log (this.firstName + ” ” + this.lastName);
};


var abhijit = new Employee () //
console.log(abhijit.fullName); // Abhijit Patel
```

fullName is set as a function, so sending it to the console would print out the function rather than the full name. All that is needed is abhijit.fullName().

Brendan

January 16, 2015 at 7:36 pm / Reply

Richard, once again thank you for such great explanations. I’m working my way through your JS roadmap using Definitive Javascript, but this provided a gre

addition.

Two questions:
– where you created the User function, what is the benefit of defining the methods I the Prototype? I understand the benefits of inheritance and read through all the comments, particularly your discussion of this topic with Jurgen. However, for this particular example, every value will be specific to the instance of User. So what is the benefit of defining in prototype, and not just adding it all to the constructor?
– on the same example, what is the rule of thumb for what should be defined in the constructor and what should be defined in the prototype? Am I right that variables are defined in the constructor and methods are defined in the prototype? Is that simplifying it too much? Is there a benefit to memory usage here?

Thanks again.

Brendan

Surender

February 23, 2015 at 1:14 am / Reply

Great write up…

Jake

February 27, 2015 at 4:25 am / Reply

Just a minor thing:

function Tree (typeOfTree) {}

should probably read

function Tree (typeOfTree) {
this.typeOfTree = typeOfTree;
}

So that the

var bananaTree = new Tree("Banana");

Does something.

Other than that, loving the site. Been messing around with JavaScript for years and only last night found your site when looking for a quick Node JS learning path. Many thanks for all the effort you put in here!

Anthony Brown

April 10, 2015 at 4:19 am / Reply

I've come to conclusion that using JS to mimic classical inheritance is overly complicated and can be achieved from Objects inheriting from other objects. A simple revealing module pattern gives us an API that is private but exposes a public API (ones that you want to be public). It's good to know all of JavaScript. If you guys haven't read the `You don't know JavaScript` books from Kyle Simpson, I suggest you do so.

Deepak

May 30, 2015 at 2:57 pm / Reply

Excellent !!! Well explained

Dhruv

June 23, 2015 at 3:19 am / Reply

Refer "Why did we manually set the copyOfParent.constructor?" under "Prototypal Inheritance by Douglas Crockford" – you have mentioned that copyOfParent doesn't have a constructor property anymore because we overwrote its entire prototype.

Now my question is, if we overwrite an object's prototype, how does it's constructor property gets lost? In many other posts on web, they overwrite

Object.prototype.constructor and not Object.constructor as you have done here. What is the difference between Object.prototype.constructor and Object.constructor?
Thanks.

Gabriel

June 23, 2015 at 3:28 pm / Reply

We have to use a constructor.

Gaurav Gupta

July 10, 2015 at 4:39 am / Reply

Hey Richard,

It was indeed a very good article for someone like me who has spent some time writing prototypes and had some doubts.

I went through the article and understood every detail of it.Now I feel like I can even explain this to someone if required.

Cheers,

pari

July 12, 2015 at 7:09 pm / Reply

what is the exact difference in writing. I am very confused about this.

Employee.prototype.firstName = “Abhijit”; AND Employee.firstName = “Abhijit”

Anil

July 30, 2015 at 1:59 am / Reply

It might help you!
OOPs In JavaScript http://goo.gl/irP6Lt

Adrian

August 6, 2015 at 8:02 pm / Reply

Hi all,

what do you think about the “Stampit” library? (https://github.com/stampit-org/stampit)

This new approach to JS Prototype is really interesting!

Buzut

August 29, 2015 at 5:45 am / Reply

Hi Richard,

Thanks for this great post. I’d like to hightlight a point. You say:

The one disadvantage of overwriting the prototype is that the constructor property no longer points to the prototype, so we have to set it manually. Hence this line:

constructor: User

“the constructor property no longer points to the prototype”, the reader might wonder –> so what?
Indeed, all the code works the same without the constructor property.

The constructor in itself has no effect on the code. It’s just in case one might need it in his logic.
http://stackoverflow.com/questions/4012998/what-it-the-significance-of-the-javascript-constructor-property

Tony

August 29, 2015 at 2:07 pm / Reply

@Adrian,

It's not new, it's how JS works to begin with, prototypes not classes. We can make JavaScript behave like a classical language because it's familiar to those coming from another language like Java or C++, or a functional language, hence the elegance and beauty of JavaScript. It's up to you how you want to use it. Most people go the classical route because it's familiar. Functional JS is becoming more and more popular do to it being easier to maintain and write, it's a very powerful way to program in general. Just my 2 cents.

raj

October 16, 2015 at 4:09 am / Reply

Thanks guys,Please keep it up!

Ryan

October 25, 2015 at 10:37 pm / Reply

I'm two years late on this post but I just wanted to say I've found it to be one of the most useful and easy to understand tutorials on OOP in javascript. Most of the time, I either find complete beginner stuff (var a = x etc) or advanced docs that assume a certain amount of foreknowledge. This was just what I needed to help bridge the gap between by beginner understanding and an intermediate level.

I'll definitely be coming back to read more. I think I have to go over those inheritance patterns a few times until the may sense to me!

Just one thing, when I ran the code, I got an error message: Uncaught ReferenceError: inheritPrototype is not defined

So I went and copied the inheritPrototype function in the example and the code worked. Was that what we were supposed to do? I thought that was just an example of the function.

Other things that would have made this tut perfect for someone as green as me:

– showing how to display the HTML output on the page rather than just the console. This is probably super easy but like I say, I'm still a beginner.

Thanks again 🙂

anil kumar

November 18, 2015 at 10:47 am / Reply

Hey,
I think if you have used childFunction and parentFunction in the explanation where you explained the parasitic inheritance in javascript,then it would have been much clear for people who are starting oops in javascript.Any ways ,i got it but it may be confusing for the new comers.

James

November 20, 2015 at 4:55 am / Reply

I know this article is rather old, but can you explain how I call inherited functionality in overridden functions?

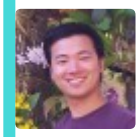
```
baseObject.prototype.doStuff = function() {
}

inheritorObject.prototype.doStuff = function() {
// how do I/what is best practice for calling baseObject.doStuff();?
// or do I simple have to call this.doStuff() and give this function a different name?
}
```

Roshith

November 25, 2015 at 1:51 am / Reply

Brilliant article. I used to write Javascript code quite a few years ago and now I wanted to understand the new concepts and techniques it uses. I came at the right place! Thanks a lot.



Anh Tran

January 26, 2016 at 12:04 am / Reply

I feel confusing when using “this” keyword for OOP. It might takes to unwanted actions of the context. What do you suggest to avoid that?

Nicholas Abrams

February 16, 2016 at 5:00 pm / Reply

You should not overwrite the prototype like you did…

SomeClass.prototype[something] = fn;

DOES NOT EQUAL

SomeClass.prototype = {
something: fn
}

The method is not the only thing inside of the prototype object which you are erasing/overwriting.

constructor: ()
__proto__: Object
These should be inside of all prototype object, but after you x.prototype = { .. } they will be gone!

Nicholas Abrams

February 16, 2016 at 5:00 pm / Reply

You should not overwrite the prototype like you did…

SomeClass.prototype[something] = fn;

DOES NOT EQUAL

SomeClass.prototype = {
something: fn
}

The method is not the only thing inside of the prototype object which you are erasing/overwriting.

constructor: ()
__proto__: Object
These should be inside of all prototype object, but after you x.prototype = { .. } they will be gone!

Beck

March 2, 2016 at 11:07 am / Reply

Essentially, we are copying all the properties and methods from the parentObject to the childObject, but we are using the copyOfParent as an intermediary for the copy. And because the childObject(x) > copyofParent prototype was overwritten during the copy, we manually set the copyOfParent constructor to the childObject. Then we set the childObject prototype to the copyOfParent so that the childObject inherits from the parentObject.

Can I modify like above?

hod caspi

March 2, 2016 at 6:16 pm / Reply

Love the way you explain and show us your knowledge to understand js better!

cheers!

Ankush

April 7, 2016 at 1:51 am / Reply

Hi , just one question regarding the constructor property you mentioned that it gets lost if prototype is overriden

```
function User(name,email){
this.name = name;
this.email = email;
this.f1 = function(){
console.log(this.name);
}
}
User.prototype = {
//constructor: User,
save:function(){
console.log("save");
}
}

var user = new User("Ankush","abc");
user.f1();
user.save();
```

Here constructor property is commented so I understand now user.constructor will point to Object but I am not able to understand what it affects on inheritance , As I am able to access both f1 and save function. Sorry if its a dumb question

RJ

November 13, 2016 at 11:05 am / Reply

Dear Sir,

Thank you for this informative documentation.

I would like just to ask with regards to the part “Implementation of Combination Constructor/Prototype Pattern” around the beginning. Because if my understanding is correct, you declared a ” constructor: User ” in he User.Prototype when you overwritten the prototype User object. I tried to test the code to understand more, and I tried removing the ” constructor: User ” and in the end I’m still able to access the prototypes methods like ” firstUser.showNameAndScores() “… Why is that happening?

Richard Bovell (Author)

March 25, 2013 at 1:48 am / Reply

HI Jeffrey, The example in this post is a real-world example. You can run the code in your browser. But I know what you mean: I could expand on the example more and build a **complete** web application that anyone can download and run on its own. I will try to do that the next time, it time permits 🙂

Richard Of Stanley (Author)

November 22, 2013 at 7:22 am / Reply

Thanks much, Aamir; I just fixed it.
Good catch.

Praga

December 15, 2013 at 6:21 am / Reply

My question is,

Private variables are not accessible inside function which are added to function prototype.

In that case, how this variables used inside prototype function?

I want explanation for that. Can we use like that?

Praga

December 15, 2013 at 3:27 pm / Reply

Could you please suggest me some good books to learn Javascript OOP concepts thoroughly?

Richard Of Stanley (Author)

January 7, 2014 at 1:05 am / Reply

Well, I take it you want to study JavaScript OOP concepts in depth. If so, you can read Chapter 6 (and reread it because it is a tough study) of Professional JavaScript for Web Developers 3rd edition. That's all you need.

Sure, there are some other OOP books, but you don't need them.

Anthony Brown

January 16, 2014 at 8:46 am / Reply

Where's the like button!

shams

March 4, 2014 at 9:26 am / Reply

Shouldn't copyOfParent.constructor = childObject; be changed to

copyOfParent.constructor = childObject.constructor;

Trackbacks for this post

1. Useful Website Articles & Tutorials | Code Chewing
2. OOP In JavaScript: What You NEED to Know | .Net Web Development | Scoop.it
3. OOP In #JavaScript : What You NEED to Know | Web Development Resources | Scoop

4. Boston web developer and web designer : Erik August Johnson : Blog
5. OOP In JavaScript: What You NEED to Know | Barış Velioğlu'nun Programlama Notları | Scoop.it
6. JavaScript | Pearltrees
7. OOP In JavaScript: What You NEED to Know | custom ecommerce | Scoop.it
8. How to Learn JavaScript Properly | JavaScript is Sexy
9. OOP In JavaScript: What You NEED to Know | Java...
10. 10 very good reasons to stop using JavaScript | LeaseWeb Labs
11. 16 JavaScript Concepts JavaScript Professionals Must Know Well | JavaScript is Sexy
12. Frontend Development « 杨明的博客
13. How to Learn JavaScript Properly - Ray Sinlao
14. Frontend Development | 杨明的博客
15. OOP In JavaScript: What You NEED to Know | The ...
16. 如何正确学习Javascript | 17khba
17. Today's Links 18 March 2015 - The New Solution News
18. 如何正确学习JavaScript — 好JSER
19. Some Kool Resources for Client Side Technologies | Insight's Delight
20. 「译」如何正确学习JavaScript | 一世浮华一场空
21. javascript学习路线 | w3croad
22. The Odin Project Progress Map | Tilly Codes
23. Daily links 2015-06-08 | Marcin Kossowski
24. Web Development Research June 11, 2015 - Angelique Roussos
25. 如何正确学习JavaScript – web研究院
26. Weekly links 2015-06-14 | Marcin Kossowski
27. 「译」如何正确学习JavaScript - YelloWish
28. 「译」如何正确学习JavaScript | 大前端
29. Starting Web Development: II – Coalworks
30. Web Dev Information Spread | Banana Phone
31. Article Spread (1) | Banana Phone
32. Javascript Weekly No.122 | ENUE Blog
33. Preparing Before Hack Reactor Begins | bash $ cat bitchblog
34. How To Learn JavaScript Properly — Introduction | My Blog
35. OOJ / Classes / Pseudoclassical etc | Daniel Moi
36. [퍼온글] 자바스크립트 배우기 - hodoogwaja
37. FRONTEND DEVELOPMENT RESOURCES BY GITHUB CONTRIBUTORS – Everyday Tips
38. FRONTEND DEVELOPMENT RESOURCES BY GITHUB CONTRIBUTORS – ElderCity

39. A Primer about Javascript's Prototype - Better Programmer
40. Understanding Javascript's Prototype - RubyEffect Blog
41. » Write asynchronous code like a boss
42. JS | Kiers McFarlane
43. Day 5 of 12h a day coding challenge – Code Support
44. Day 5 of 60 days coding challenge - Code Support
45. Objektorientierung in JavaScript | techscouting through the java news
46. OOPS in Javascript | Lamp Stack

Leave a Reply

Name | Email (hidden) | Website

Comment:

Submit Comment

Notify me of follow-up comments by email.

Notify me of new posts by email.

 JavaScript is Sexy About Contact Archive