JavaScript Object Prototypes

All JavaScript objects inherit properties and methods from a prototype.

In the previous chapter we learned how to use an **object constructor**:

### **Example**

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Objects</h2>

<p id="demo"></p>

<script>

function Person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

}

var myFather = new Person("John", "Doe", 50, "blue");

var myMother = new Person("Sally", "Rally", 48, "green");

document.getElementById("demo").innerHTML =

"My father is " + myFather.age + ". My mother is " + myMother.age;

</script>

</body>

</html>

We also learned that you can **not** add a new property to an existing object constructor:

### **Example**

### <!DOCTYPE html>

### <html>

### <body>

### <h2>JavaScript Objects</h2>

### <p>You cannot add a new property to a constructor function.</p>

### <p id="demo"></p>

### <script>

### function Person(first, last, age, eye) {

### this.firstName = first;

### this.lastName = last;

### this.age = age;

### this.eyeColor = eye;

### }

### Person.nationality = "English";

### var myFather = new Person("John", "Doe", 50, "blue");

### var myMother = new Person("Sally", "Rally", 48, "green");

### document.getElementById("demo").innerHTML =

### "The nationality of my father is " + myFather.nationality;

### </script>

### </body>

</html>

To add a new property to a constructor, you must add it to the constructor function:

### **Example**

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Objects</h2>

<p id="demo"></p>

<script>

function Person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

this.nationality = "English";

}

var myFather = new Person("John", "Doe", 50, "blue");

var myMother = new Person("Sally", "Rally", 48, "green");

document.getElementById("demo").innerHTML =

"The nationality of my father is " + myFather.nationality + ". The nationality of my mother is " + myMother.nationality;

</script>

</body>

</html>

# JavaScript Closures

JavaScript variables can belong to the **local** or **global** scope.

Global variables can be made local (private) with **closures**.

## **Global Variables**

A function can access all variables defined **inside** the function, like this:

<!DOCTYPE html>

<html>

<body>

<p>A function can access variables defined inside the function:</p>

<button type="button" onclick="myFunction()">Click Me!</button>

<p id="demo"></p>

<script>

function myFunction() {

var a = 4;

document.getElementById("demo").innerHTML = a \* a;

}

</script>

</body>

</html>

But a function can also access variables defined **outside** the function, like this:

<!DOCTYPE html>

<html>

<body>

<p>A function can access variables defined outside the function:</p>

<button type="button" onclick="myFunction()">Click Me!</button>

<p id="demo"></p>

<script>

var a = 4;

function myFunction() {

document.getElementById("demo").innerHTML = a \* a;

}

</script>

</body>

</html>

In the last example, **a** is a **global** variable.

In a web page, global variables belong to the window object.

Global variables can be used (and changed) by all scripts in the page (and in the window).

In the first example, **a** is a **local** variable.

A local variable can only be used inside the function where it is defined. It is hidden from other functions and other scripting code.

Global and local variables with the same name are different variables. Modifying one, does not modify the other.

Variables created **without** the keyword **var**, are always global, even if they are created inside a function.

## **Variable Lifetime**

Global v-ariables live as long as your application (your window / your web page) lives.

Local variables have short lives. They are created when the function is invoked, and deleted when the function is finished.

## **A Counter Dilemma**

Suppose you want to use a variable for counting something, and you want this counter to be available to all functions.

You could use a global variable, and a function to increase the counter:

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Function Closures</h2>

<p>Counting with a global variable.</p>

<p id="demo"></p>

<script>

// Initiate counter

var counter = 0;

// Function to increment counter

function add() {

counter += 1;

}

// Call add() 3 times

add();

add();

add();

// The counter should now be 3

document.getElementById("demo").innerHTML = "The counter is: " + counter;

</script>

</body>

</html>

There is a problem with the solution above: Any code on the page can change the counter, without calling add().

The counter should be local to the add() function, to prevent other code from changing it:

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Function Closures</h2>

<p>Counting with a local variable.</p>

<p id="demo"></p>

<script>

// Initiate counter

var counter = 0;

// Function to increment counter

function add() {

var counter = 0;

counter += 1;

}

// Call add() 3 times

add();

add();

add();

// The result is not 3 because you mix up the globaland local counter

document.getElementById("demo").innerHTML = "The counter is: " + counter;

</script>

</body>

</html>

It did not work because we display the global counter instead of the local counter.

We can remove the global counter and access the local counter by letting the function return it:

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Closures</h2>

<p>Counting with a local variable.</p>

<button type="button" onclick="myFunction()">Count!</button>

<p id="demo">0</p>

<script>

// Function to increment counter

function add() {

var counter = 0;

counter += 1;

return counter;

}

// Trying to increment the counter

function myFunction(){

document.getElementById("demo").innerHTML = add();

}

</script>

</body>

</html>

It did not work because we reset the local counter every time we call the function.

**A JavaScript inner function can solve this.**

## **JavaScript Nested Functions**

All functions have access to the global scope.

In fact, in JavaScript, all functions have access to the scope "above" them.

JavaScript supports nested functions. Nested functions have access to the scope "above" them.

In this example, the inner function plus() has access to the counter variable in the parent function:

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Function Closures</h2>

<p>Counting with a local variable.</p>

<p id="demo">0</p>

<script>

document.getElementById("demo").innerHTML = add();

function add() {

var counter = 0;

function plus() {counter += 1;}

plus();

return counter;

}

</script>

</body>

</html>

This could have solved the counter dilemma, if we could reach the plus() function from the outside.

We also need to find a way to execute counter = 0 only once.

**We need a closure.**

## **JavaScript Closures**

Remember self-invoking functions? What does this function do?

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Closures</h2>

<p>Counting with a local variable.</p>

<button type="button" onclick="myFunction()">Count!</button>

<p id="demo">0</p>

<script>

var add = (function () {

var counter = 0;

return function () {counter += 1; return counter;}

})();

function myFunction(){

document.getElementById("demo").innerHTML = add();

}

</script>

</body>

</html>

## **Example Explained**

The variable add is assigned the return value of a self-invoking function.

The self-invoking function only runs once. It sets the counter to zero (0), and returns a function expression.

This way add becomes a function. The "wonderful" part is that it can access the counter in the parent scope.

This is called a JavaScript **closure.** It makes it possible for a function to have "**private**" variables.

The counter is protected by the scope of the anonymous function, and can only be changed using the add function.

A closure is a function having access to the parent scope, even after the parent function has closed.

# JavaScript Object Methods

<!DOCTYPE html>

<html>

<body>

<h2>The JavaScript <b>this</b> Keyword</h2>

<p>In this example, <b>this</b> represents the <b>person</b> object.</p>

<p>Because the person object "owns" the fullName method.</p>

<p id="demo"></p>

<script>

// Create an object:

var person = {

firstName: "John",

lastName : "Doe",

id : 5566,

fullName : function() {

return this.firstName + " " + this.lastName;

}

};

// Display data from the object:

document.getElementById("demo").innerHTML = person.fullName();

</script>

</body>

</html>

## **The**this**Keyword**

In a function definition, this refers to the "owner" of the function.

In the example above, this is the **person object** that "owns" the **fullName** function.

In other words, **this.firstName** means the **firstName** property of **this object**.

Read more about the this keyword at [JS this Keyword](https://www.w3schools.com/js/js_this.asp).

## **JavaScript Methods**

JavaScript methods are actions that can be performed on objects.

A JavaScript **method** is a property containing a **function definition**.

|  |  |
| --- | --- |
| **Property** | **Value** |
| firstName | John |
| lastName | Doe |
| Age | 50 |
| eyeColor | blue |
| fullName | function() {return this.firstName + " " + this.lastName;} |

Methods are functions stored as object properties.

## **Accessing Object Methods**

You access an object method with the following syntax:

*objectName.methodName()*

You will typically describe fullName() as a method of the person object, and fullName as a property.

The fullName property will execute (as a function) when it is invoked with ().

This example accesses the fullName() **method** of a person object:

<!DOCTYPE html>

<html>

<body>

<p>Creating and using an object method.</p>

<p>A method is actually a function definition stored as a property value.</p>

<p id="demo"></p>

<script>

var person = {

firstName: "John",

lastName : "Doe",

id : 5566,

fullName : function() {

return this.firstName + " " + this.lastName;

}

};

document.getElementById("demo").innerHTML = person.fullName();

</script>

</body>

</html>

If you access the fullName **property**, without (), it will return the **function definition**:

## **Using Built-In Methods**

This example uses the toUpperCase() method of the String object, to convert a text to uppercase:

var message = "Hello world!";  
var x = message.toUpperCase();

The value of x, after execution of the code above will be:

HELLO WORLD!

## **Adding a Method to an Object**

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Object Constructors</h2>

<p id="demo"></p>

<script>

// Constructor function for Person objects

function Person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

}

// Create 2 Person objects

var myFather = new Person("John", "Doe", 50, "blue");

var myMother = new Person("Sally", "Rally", 48, "green");

// Add a name method to first object

myFather.name = function() {

return this.firstName + " " + this.lastName;

};

// Display full name

document.getElementById("demo").innerHTML =

"My father is " + myFather.name();

</script>

</body>

</html>

# JavaScript — Inheritance

### What is Inheritance

Inheritance in most class-based object-oriented languages is a mechanism in which one object acquires all the properties and behaviors of another object. JavaScript is not a class-based language although class keyword is introduced in ES2015, it is just syntactical layer. JavaScript still works on prototypechain.

### Classical Inheritance vs Prototypal Inheritance

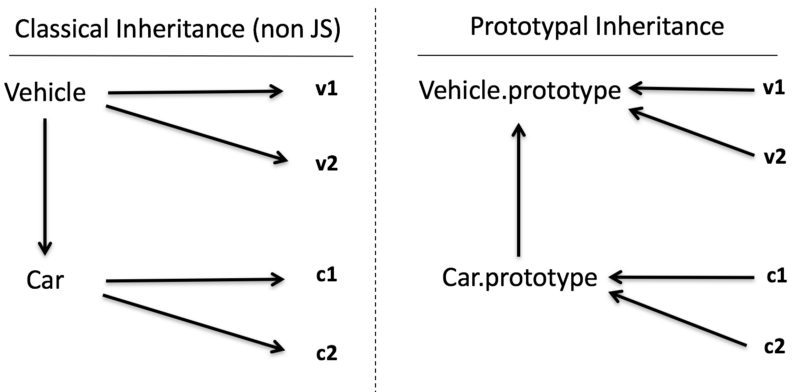


diagram of inheritance

#### Classical Inheritance (non-javascript)

* Vehicle is parent class and v1,v2 are the instances of Vehicle
* Car is child class of Vehicle and c1, c2 are instances of Car
* In classical inheritance it creates a copy of the behavior from parent class into the child when we extend the class and after that parent and child class are separate entity.
* Similarly, another copy of the behavior happens when we create an instance or object out of the class and both are separate entity again.
* It’s like car is manufactured from the vehicle and car blueprints but after that both are separate entity because they are not linked because It’s a copy. That’s the reason all arrows going downwards (property and behavior flowing down)

#### Prototypal Inheritance (Behavior delegation pattern)

* v1 and v2 are linked to Vehicle.prototype because it’s been created using new keyword.
* Similarly, c1 and c2 is linked to Car.prototype and Car.prototype is linked to Vehicle.prototype.
* In JavaScript when we create the object it does not copy the properties or behavior, it creates a link. Similar kind of linkage gets created in case of extend of class as well.
* All arrows go in the opposite direction compare to classical non-js inheritance because it’s a behavior delegation link. These links are known as prototype chain.
* This pattern is called Behavior Delegation Pattern which commonly known as **prototypal inheritance**in JavaScript.

*You may go through article*[*JavaScript — Prototype*](https://codeburst.io/javascript-prototype-cb29d82b8809)*to understand the****prototypechain****in depth.*

#### Example of prototypal inheritance

* Usage of Object.create() to achieve classical inheritance.
* In the below code snippet, Car.prototype and Vehicle.prototype is linked with help of Object.create() function.

**// Vehicle - superclass**function Vehicle(name) {  
 this.name = name;  
}

**// superclass method**  
Vehicle.prototype.start = function() {  
 return "engine of "+this.name + " starting...";  
};

**// Car - subclass**  
function Car(name) {  
 Vehicle.call(this,name); **// call super constructor.**  
}

**// subclass extends superclass**  
Car.prototype = Object.create(Vehicle.prototype);

**// subclass method**Car.prototype.run = function() {  
 console.log("Hello "+ this.start());  
};

**// instances of subclass**  
var c1 = new Car("Fiesta");  
var c2 = new Car("Baleno");

**// accessing the subclass method which internally access superclass method**  
c1.run(); // "Hello engine of Fiesta starting..."  
c2.run(); // "Hello engine of Baleno starting..."

* In the above code, object c1 gets access to method run() and method start() because of below prototype chain. As per below diagram, we can see object c1 does not have these methods but it has links to go upwards.
* keyword this in above code is nothing but current execution context of each method which is c1 and c2.

*To understand keyword****this****in details, you can go through article*[*JavaScript — All about this and new keyword*](https://codeburst.io/all-about-this-and-new-keywords-in-javascript-38039f71780c)*.*

Diagrammatic representation of above code

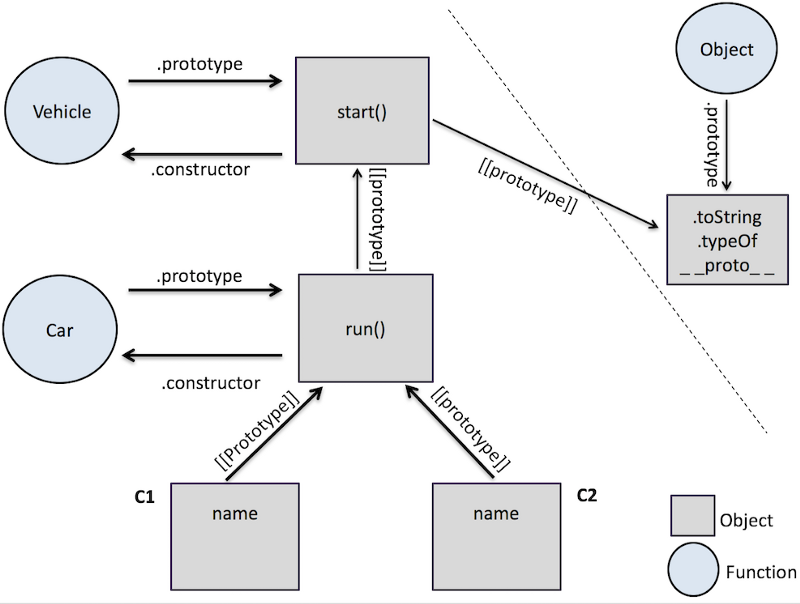


diagram of javascript inheritance

#### Objects linked to other objects

* Now we will simplify the previous example code of inheritance with focusing only on objects and object linkages.
* So we will try to remove .prototype, constructor and new keyword, will be thinking about only objects.
* We will be using Object.create()function to create all the linkages between objects.

Below is simplified code of previous example.

**// base object with methods including initialization**  
var Vehicle = {  
 init: function(name) {  
 this.name = name;  
 },  
 start: function() {  
 return "engine of "+this.name + " starting...";  
 }  
}

**// delegation link created between sub object and base object**var Car = Object.create(Vehicle);

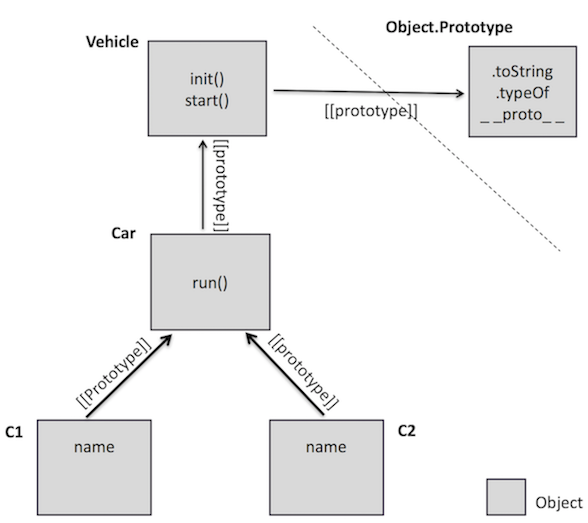
**// sub object method**  
Car.run = function() {  
 console.log("Hello "+ this.start());  
};

**// instance object with delegation link point to sub object**  
var c1 = Object.create(Car);  
c1.init('Fiesta');

var c2 = Object.create(Car);  
c2.init('Baleno');

c1.run(); // "Hello engine of Fiesta starting..."  
c2.run(); // "Hello engine of Baleno starting..."

Diagrammatic representation of above code



objects linking

* We can see now, how we have removed the complexity of new, all the .prototype, constructor functions and call method everything and still achieved the same result.
* Only thing which matters is object c1 linked to another object and then linked to another object again and so on.
* This is called object delegation pattern as well.

## **ECMAScript 2015**

ES6, also known as ECMAScript2015, introduced classes.

A class is a type of function, but instead of using the keyword function to initiate it, we use the keyword class, and the properties are assigned inside a constructor() method.

## **Class Definition**

Use the keyword class to create a class, and always add the constructor() method.

The constructor method is called each time the class object is initialized.

### **Example**

A simple class definition for a class named "Car":

class Car {  
  constructor(brand) {  
    this.carname = brand;  
  }  
}

Now you can create objects using the Car class:

### **Example**

Create an object called "mycar" based on the Car class:

class Car {  
  constructor(brand) {  
    this.carname = brand;  
  }  
}  
mycar = new Car("Ford");

**Note:** The constructor method is called automatically when the object is initialized.

## **Methods**

The constructor method is special, it is where you initialize properties, it is called automatically when a class is initiated, and it has to have the exact name "constructor", in fact, if you do not have a constructor method, JavaScript will add an invisible and empty constructor method.

You are also free to make your own methods, the syntax should be familiar:

### **Example**

Create a method named "present":

class Car {  
  constructor(brand) {  
    this.carname = brand;  
  }  
  present() {  
    return "I have a " + this.carname;  
  }  
}  
  
mycar = new Car("Ford");  
document.getElementById("demo").innerHTML = mycar.present();

As you can see in the example above, you call the method by referring to the object's method name followed by parentheses (any parameters would go inside the parentheses).

### **Example**

Send a parameter to the "present()" method:

class Car {  
  constructor(brand) {  
    this.carname = brand;  
  }  
  present(x) {  
    return x + ", I have a " + this.carname;  
  }  
}  
  
mycar = new Car("Ford");  
document.getElementById("demo").innerHTML = mycar.present("Hello");