TECHNICAL UNIVERSITY OF MOLDOVA

ANALYSIS AND DESIGN OF OBJECT ORIENTED PROGRAMS

Laboratory Work nr. 1

Autor:
Nicușor CHICIUC

Controlor:
Vlad DRAHNEA

February 15, 2016

1 Introduction

Even mentioning that JavaScript is an OOP language can provoke a big debate in the programming circle. Some would even go as far as blame you for calling it a *programming language* and not a *scripting language* (even though scripting languages are a subset of programming languages). In this article I will not try to start a flame war or to condemn other people's opinion about aspects of the current programming culture. Instead, I'll try to show the reader some interesting things that can be done with JavaScript.

Of course the article will not focus solely on some cool tricks, it'll try to provide the reader with an understanding framework on which he can create new and innovative ways in which to program and think in JavaScript.

1.1 A gentle introduction

The *problem* with JavaScript is that it doesn't have classes, like usual OOP languages. Of course you have the `new` keyword which seems like calling a class to create a new object but it's not so straightforward. I'll talk about the `new` keyword later but now let's start with some simple objects.

The easiest way to create a new object is like this:

```
var newObject = {}
newObject.property01 = 'some property'
newObject.otherProperty = 40 + 2
```

You don't even need a semicolon, how cool is that. In this case you can't even talk about OOP. There's no polymorphism going on, no inheritance (actually you inherit the basic `Object` prototype) and not even encapsulation. This object is basically a structure that has 2 attributes. But we're on the right track.

If you're coming from an OOP world you expect to be able to have a blueprint for objects, to be able to create the same type of object over and over again, as much as your heart desires. In the case above, you can't do that.

JavaScript provides such a mechanism. It's the so called *constructor function*. There isn't any difference in the way you define it from a simple function, the difference is how you use it. The next example shows it in action:

```
var newObject = new SuperClass()
var newObject02 = new SuperClass()

console.log(newObject.property01)       // prints: some property
console.log(newObject02.otherProperty)  // prints: 42

function SuperClass () {
    this.property01 = 'some property'
    this.otherProperty = 40 + 2
}
```

We have the `this` keyword and the `new` keyword, it looks like something OOP-like is going on. Let's get a simpler example and take everything step by step.

```
function Car () {}

var c0 = new Car()
var c1 = Car()

console.log(c0) // prints: {}
console.log(c1) // prints: undefined
```

You don't get any errors for calling the function `Car` without the `new` keyword, it just doesn't create an object for you. It's invoked as a simple function. What happens is that the `new` keyword creates a new object (just like we did with {}) and sets it as the context to the `Car` function.

Before we go forward we have to discuss the `prototype` and `constructor` attribute. And before that we have to understand that in JavaScript (mostly) everything is an object.

2 Objects in Javascript

Just as you can assign an attribute to normal object you can also assign an attribute to a function, or to and array, or a number etc. Here's an example:

```
var arr = [1, 2, 3, 'cat', 5]
arr.someInfo = 'no info for you'

for(var i = 0; i < arr.length; i++)
    console.log(arr[i])   // 1, 2, 3, 'cat', 5
console.log(arr.someInfo) // 'no info for you'

function getTheCat () {
    return 'cat'
}
getTheCat.sound = 'meow'

console.log(getTheCat())     // 'cat'
console.log(getTheCat.sound) // 'meow'
```

These are some attributes we just inserted there for the sake of it but the truth is that there are some default attributes that are constantly used but usually remain unknown. Of course the fact that you can do a lot of cool stuff in JavaScript doesn't mean you should always do it. But in some scenarios this knowledge can come in handy.

2.1 Prototype and Constructor

2.1.1 prototype

Every function in JavaScript has a `prototype` property that initially references to an empty object. This property doesn't have an important role until you use the function as a constructor. Let's return to the `Car` example and add a function to the prototype.

```
function Car () {}

Car.prototype.sound = function () {
    return 'wrrrrom'
}

var c0 = new Car()

console.log(c0.sound()) // 'wrrrrom'
```

Even though the `sound` function is not a property of the `c0` object it can be invoked through it. What this means is that the `prototype` serves as something like a blueprint for the newly created object.

Note that the object wasn't changed, no property was added to it. When you call a function in JavaScript, the interpreter checks if the object has such a property, if it doesn't, the interpreter goes to the prototype of the function that created the object and checks if there's a property like that there. This is called a *prototype chain*.

2.1.2 constructor

The `constructor` property points to the function that created the current object. Here's an example:

```
function A(){}

var a = new A()

console.log(a.constructor === A)      // true
console.log(a.constructor === Object) // false
```

Even though `a` has `Object` in it's prototype chain it is not its direct constructor, that's why you'll get false on the second log.

2.1.3 instanceof

In order to check whether an object is an instace of a constructor function one can use the `instanceof` keyword. It does this by going through all the prototype chain of the object and checks whether the function is equal to any of the constructors it finds.

```
function A(){}

var a = new A()
var b = {}

console.log(a instanceof A)      // true
console.log(a instanceof Object) // true
console.log(b instanceof A)      // false
console.log(b instanceof Object) // true
```

You can see that in this case `a` is actually an instance of `Object`, but it's not the first contructor in it's prototype chain.

3 Instance properties and prototype properties

As you've seen before an object can have its own properties or it can refer to properties in its prototype chain. There are some question that arise. What if there is the same function in the prototype of the object and in his instance? Can a prototype access instance properties of the object?

```
function A () {
    this.instProp = 'THIS. IS. INSTANCE!!!'

    this.getSome = function () {
        return 'from the instance'
    }
}

A.prototype.getSome = function () {
    return 'from the prototype'
}

A.prototype.getOther = function () {
    return this.instProp
}


var a = new A()


console.log(a.getSome())  // 'from the instance'

console.log(a.getOther()) // 'THIS. IS. INSTANCE!!!'
```

As you can see, the closest property in the prototype chain is used if it's called. Even though a method is in the property of the object it will still refer to the property in the object itself.

```
function A () {
    this.prop = 'THIS. IS. INSTANCE!!!'
}

A.prototype.prop = 'THIS. IS. PROTOTYPE!!!'

A.prototype.getOther = function () {
    return this.prop
}

var a = new A()

console.log(a.getOther())  // 'from the instance'
```

Even though the prototype has its own `prop` property and even if we use the `this` keyword we will still refer to the property nearest to the object from which the method was invoked. If we remove the property from the function it will refer its own property.

```
function A () {}

A.prototype.prop = 'THIS. IS. PROTOTYPE!!!'

A.prototype.getOther = function () {
    return this.prop
}

var a = new A()

console.log(a.getOther())  // 'THIS. IS. PROTOTYPE!!!'
```

As you've seen till now, the JavaScript language is extremely flexible, compared to other OOP languages. This flexibility of course comes at the price of more resources used, especially since most JavaScript implementations are interpreters. But we're not programming microcontrollrs now so don't think about it too much.

4 Inheritance using the prototype chain

As we've seen earlier, we can somehow mimic inheritance by using the prototype. Suppose we have a class `Car` and we want to create a subclass `Motorcycle` that'll have the same methods as `Car`. We can do this by copying the methods from the prototype of the Car to the prototype of the Motorcycle.

```
function Car () {}
Car.prototype.sound = function () {
    return 'wrrrrom'
}

function Motorcycle () {}
Motorcycle.prototype.sound = Car.prototype.sound;


var c0 = new Motorcycle()

console.log(c0.sound()) // 'wrrrrom'
console.log(c0 instanceof Car) // false
```

Running the script shows that we indeed made the motorcycle to have a nice sound but this is not how inheritance should be done. This is basically copying. We can of course do this for all the properties of a object but there should be other methods.

One of the best techniques to achieve this is to make a prototype use a new instance of the inhereted class.

```
function Car () {}
Car.prototype.sound = function () {
    return 'wrrrrom'
}

function Motorcycle () {}
Motorcycle.prototype = new Car()


var c0 = new Motorcycle()

console.log(c0.sound()) // 'wrrrrom'
console.log(c0 instanceof Car) // true
```

Another technique that you might think about is to do it like this:

```
Motorcycle.prototype = Car.prototype
```

The problem with this is that every change in the prototype of the motorcycle will change the prototype of the Car and this is not desired.

Every native JavaScript object constructor (Array, Function, Object, Number, etc.) have prototype properties that can be changed, added or removed. This make sense and can prove itself as an interesting way to manipulate the language itself. Yet a lot of code relies on native properties so doing so is very error-prone.

Below is a method that can be used to delete any DOM element.

```
HTMLElement.prototype.remove = function () {
    if (this.parentNode)
        this.parentNode.removeChild(this)
}
```

That's it, now the `remove` method will be available to all DOM elements.

4.0.4 hasOwnProperty

Even though the prototype chain is a cool feature, oftentimes we need to use just the properties that are instance properties. To do this, there is the `hasOwnProperty` method. It returns true if the property passed as an argument is defined in the instance of the object and false otherwise.

```
function Car () {}
Car.prototype.sound = function () {
    return 'wrrrrom'
}

var c0 = new Car()

console.log(c0.hasOwnProperty('sound'))                       // 'false'
console.log(c0.constructor.prototype.hasOwnProperty('sound')) // true
```

4.0.5 Real class-like syntax

While it's good that JavaScript lets us some kind of inheritance via prototypes, most might like to have a more standard syntax to deal with all the flexibility of JavaScript. There should be a system that would trivialize the creation of constructor functions and prototypes, a more sane way to do prototype inheritance, etc.

There are a number of JavaScript libraries that allow some classic OOP traits to be implemented in JavaScript (base2, Prototype, ease.JavaScript, etc.). Creating a fully usable code is not a joke and requires good knowledge and a lot of time for testing and stuff. If you got to this article so far you probably have some ideas of your own on have to make inheritance work so you can try your own solutions.

5 Encapsulation

Another important part of OOP is that objects cannot directly manipulate the state of other objects, the so called *Encapsulation Principle*. As you've know, JavaScript doesn't have private variables. So what's the way to do this? If you thought about closures you were correct. And if you don't know what closures are keep reading.

5.1 Closures

Closures are a very important part of JavaScript. Without them there wouldn't even be JavaScript. Let's start simple.

```
var outer = 'outerValue'

var later

function outerFunction () {
    var inner = 'innerValue'

    function innerFunction () {
        console.log(inner)
        console.log(outer)
    }

    later = innerFunction
}


//console.log(inner)  // will output an error
console.log(outer)    // 'outerValue'

outerFunction()

later() // 'innerValue'
        // 'outerValue'
```

Of course, most of us know about scope. For example the variable `inner` cannot be accesed outside of the function `outerFunction` and will probably die after the invocation of the function. Yet as we see in the call of `later` function we actually have access to both the inner and the outer variables.

How does this happen, why do those variable even exit? The answer is simple, closure. When we declared the `innerFunction` inside the `outerFunction` not only was the function declaration defined, but also a closure was created that encompasses not only the function declaration itself but also all the variables that are in scope at the point of the declaration.

Even if `innerFunction` was executed when its scope goes away it still has access to the original scope in which it was declared. This is what closures are about. They create something like a *safety bubble* of the function and the variables that are in scope at the point of the function's declaration so that the function will be able to execute.

5.1.1 Self invoking functions

Before we go to the next step I'll say few words about self-invoking functions. These are functions that are invoked in the place where they are created.

```
(function(arg){
    console.log(arg) // 'some'
})('some');

// or

(function(){

}())
```

They are useful in conjunction with closures as I will show in the next paragraphs.

5.2 Usage

Closures can be used in conjunction with constructor classes in order to provide some type of OOP encapsulation and inheritance. Check the following example:

```
var Car = (function () {
    var privateStaticVariable = 'something'
    var privateStaticFunction = function () {}

    function _Car () {
        var privateProperty = 'lorem ipsum'

        this.instanceVariable = 'whatever'
    }

    _Car.prototype = new SomeUpperClass()
    _Car.prototype.anotherFunction = function () {}

    return _Car
})()
```

As you can see, inside an self-invoking function we declare a constructor function and return it. Because the function declared a closures, all the local variables can be seen from other methods of the object yet cannot be accessed from the outside. At the same time I set up the prototype chain. Because everything is inside a self-invoking function we also have the advantage to fold the code and view it as the actual blueprint of the object. Note that the code inside is executed only once and the user has access just to the Car function which is a constructor. This is a very cool way to do OOP in JavaScript.

6 Conclusion

Even though this article didn't provide some kind of cookbook recipes for implementing production-ready OOP in JavaScript, I hope it showed the reader that it's indeed possible to do it. But the most important thing that I wanted to prove was that JavaScript is a very flexible and versatile language. Thank you for reading.

References

[1] John Reisig, *Secrets of The JavaScript Ninja*

[2] Mozilla Developer Network, *Introduction to Object-Oriented JavaScript*