# Introduction to Prototypes in JavaScript

Tobias Wohlers (394689)
Leon Spitzer (394495)
Supervisor: D. Korzeniewski

May 9, 2019

## Introduction

Most developers associate Java or C-languages with a typical object oriented programming language. JavaScript is not a typical object oriented programming language. In the first section we want to discuss differences between the Java object model and the object model of JavaScript. The principle of constructs and how they work in JavaScript will be explained in the second section. The third section is about prototypes, what exactly they are, what they do and how they work. Also, we will explain how object inheritance works with prototypes.

## 1 Object model

First of all, JavaScript is not like Java. Both of them have objects, but JavaScript has no classes. While this may confuse some, coming from a language like Java or C++, it can make programming very flexible. The understanding of how objects work is very different to Java.

### 1.1 What are objects?

The main difference between Java and JavaScript is that in Java methods and attributes exist as properties on objects. Attributes could be primitive values like integers, strings or booleans just like objects. In JavaScript everything is an object, including primitive values and methods. This gives the possibility to assign a function to a variable and move them across to different variables like example 1 shows.

```
var myFunction = function() {
    console.log("Starting engines");
}
```

```
var mySecondFunction = myFunction;
mySecondFunction();                        //"Starting engines"
```

Example 1: Functions act like objects

The first line creates a new variable and assigns a function object to it. The second line creates a second variable and copies the function object from `myFunction` into `mySecondFunction`. In the last line the function is getting called and prints `"Starting engines"` in to the console. So we can be sure that we always have objects, and we need no kind of conversion to assign properties with values. But on other hand we have to check what kind of object the property is. This can be a benefit, for example in a small application because we don‘t need to worry about what kind of property we have. But thus it can be a disadvantage for larger applications because we have to check what kind of object it is to handle the object correctly.

## 1.2 Creating objects

In Java we need a class to create objects. Classes are like a blueprint for objects in which all attributes (which are the data of the object) and methods (which give the object the functionality) are defined. Some functionality can be defined during the construction of an object in the constructor method.

In JavaScript however we can create an object without any blueprints. We can create a default object and add properties to it. This example shows a simple object creation.

```
var myObject = new Object;
myObject.foo = 123;
myObject.fooFunction = myFunction;
```

Example 2: Creating a new default object

The first line creates a new object `myObject` and the second and third line add properties to the object. This is also a main difference between Java and JavaScript. In Java we can’t add or remove any attributes or functions after the creation of an object. In JavaScript however it is possible to add a property to an object at any time.

The biggest advantage of this is that we can make small object oriented applications without creating any overhead and without maintaining any kind of class structure. This gives the possibility that programmers, which are used to typical object oriented languages, like Java, have fewer problems to understand the applications. It is also possible to change a small detail without the trouble of checking all classes. This is a big advantage in large applications. But on the other hand it has a big disadvantage. It could be difficult to keep the overview what an object can do and what properties it has. This can be solved with a strict guideline how objects should be created.

But it is still possible to define a blueprint like a class with constructors in JavaScript. How constructors work in JavaScript will be the topic of the next section.

### 1.3 Frozen objects

In object-oriented programming exists the notion that there are certain elements that are not intended to be configured or extended outside their current context. In some languages you can add the final keyword to achieve this effect. In JavaScript you can use for example the method `Object.freeze()` to manipulate the changeability of properties from objects. When using the method `Object.freeze()` on an object, you can't add or remove properties after. Neither can you change property types or write to any data properties. All data types of that object will forever be set to read-only. Implementing features like this would be useful when changing certain object properties leads to bad behavior elsewhere in the application. For further understanding serves the next example.

```
1  console.log(rocket1.name);               //"Astro12"
2  console.log(Object.isFrozen(rocket1));   //false
3
4  Object.freeze(rocket1);
5
6  rocket1.name = "Ariane";
7  console.log(rocket1.name);               //"Astro12"
8  console.log(Object.isFrozen(rocket1));   //true
9
10 delete rocket1.name;
11 console.log(rocket1.name);               //"Astro12"
```

Example 3: Effects of Object.freeze() on rocket1

In this example `rocket1` is frozen. The method `Object.isFrozen()` returns if an object is frozen or not. ④ This line freezes `rocket1` which will affect the changeability of it. ⑦ Trying to change the name property has no effect on the frozen `rocket1`. ⑩ Nor can the property be deleted. You can see that it is quite simple to set the configurability of objects. Having this kind of flexibility adds another aspect to the already very dynamic programming style of JavaScript.

## 2 Constructors

In constructors we can define default properties like functions or data objects. Example 3 shows a simple constructor.

```
1 function Rocket(name) {
2     this.name = name;
3 }
```

Example 4: Simple constructor

The first line defines a new constructor function which is a function started with a capitalized letter and the arguments that the function takes. The second line sets a new property called `name` to the object. Now we can create an object like example 4 shows.

```
var myRocket = new Rocket("Ariane");
console.log(myRocket.name);              //"Ariane"
```

Example 5: Creating an object from constructor

In this example the first line creates a new `Rocket` object with the name `"Astro12"`. In the second line the name is printed in to the console. We can also add functions to objects to get some standardized functionality to it.

```
function Rocket(name) {
    this.name = name;
    this.sayName = function() {
        console.log(this.name);
    }
};
```

Example 6: Constructor with a function

With this new constructor we can create different objects and all of them know the function `sayName`, for example.

```
var rocket1 = new Rocket("Astro12");
var rocket2 = new Rocket("Apollo11");
rocket1.sayName();                       // logs "Astro12" to the console
rocket2.sayName();                       // logs "Apollo11" to the console
```

Example 7: Creating two objects of same constructor

The first and second line create different objects with the same constructor. In the third and fourth line we call the standardized function `sayName`.

With constructors we have the possibility to create a class like structure. This can give the advantage of a better overview, what objects can do and what properties they have. In larger applications this can be an advantage because everyone can see what an object can do and everyone can always create the same object with the same properties if needed. A second advantage is that we have less code duplication, if we have a good guideline where we save all constructors and where we can find them. On the other hand if not everyone clean up the constructors after removing their usage it could be possible that we have unused constructors which enhances the code base and decreases the readability of the code in larger applications.

## 3 Prototypes

In JavaScript everything that has to do with inheritance will work with the so-called prototypes. First, we need to clarify what a prototype is and then we'll discuss the functionality and usage in JavaScript.

## 3.1 What is a Prototype?

A prototype isn't very different from a normal object. The special thing about the prototype object is, that it is getting referenced by other objects. Every Object in JavaScript has a [[Prototype]] attribute which points to a specific prototype (with some few exceptions).

This object will inherit all properties of that prototype. Objects created with the same constructor reference the same prototype. That means that every object created with the same constructor has access to methods and properties on the shared prototype. Generally, you define shared methods and primitive value properties on prototypes. When having a big data structure, it is essential to outsource repetitive code to the prototype to minimize code redundancy. To show the basic concept of a prototype structure we use the following example.

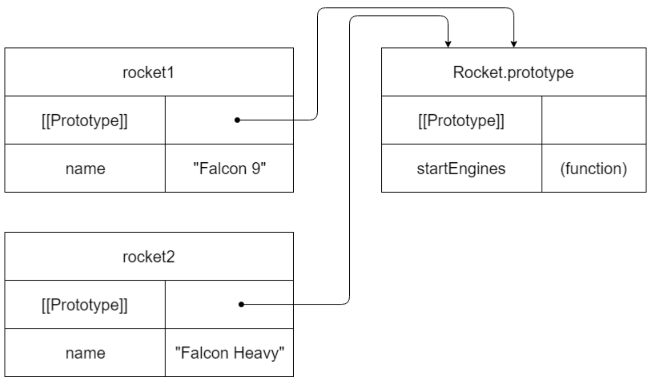

Figure 1: `[[Prototype]]` property of `rocket1` and `rocket2` point to the same prototype

```
1 function Rocket(name){
2     this.name = name;
3 }
4 Rocket.prototype.startEngines = function(){
5     console.log("Starting engines");
6 }
7 var rocket1 = new Rocket("Falcon 9");
8 var rocket2 = new Rocket("Falcon Heavy");
```

Example 8: `rocket1` and `rocket2` are created with the Rocket constructor

①: This is the Rocket constructor which automatically creates a prototype named `Rocket.prototype`. We can see in objects of Rocket that the internal property `[[Prototype]]` keeps track on which prototype to reference. ④ The same way we can add and remove properties from objects, we can add and remove properties from the prototype (because it is just an object). Because `rocket1` and `rocket2` are created from the same constructor they refer to the same prototype, which reduces code duplication because `startEngines()` doesn't need to exist on both `rocket1` and `rocket2`. So if we want to share methods with every object of an object-type it is much more efficient to put the methods on the prototype rather than to copy and paste it in every object. When calling properties we get the following results.

```
1 console.log(rocket1.name);                //"Falcon 9";
2 console.log(rocket2.name);                //"Falcon Heavy";
3 rocket1.startEngines();                   //"Starting engines";
4 rocket2.startEngines();                   //"Starting engines";
```

Example 9: `rocket1` and `rocket2` share the method `startEngines()`

①,②: `rocket1.name` and `rocket2.name` is defined on the Rocket object itself, so there is no new search process involved here.

③ But when using the method `startEngines()` on the Object `rocket1`, the JavaScript engine cannot find an internal property of that name on `rocket1`. After searching for an internal property on `rocket1` the prototype of `rocket1, Rocket.prototype` is searched for a property that name.

Now the property is found and can be used to print `"Starting engines"` into the console. Without implementing the function on the object itself it still can be used because its defined on the prototype. If the JavaScript engine wouldn't find a property that name on the object nor on one of its prototypes, undefined would be returned.

It is important to understand that the prototype object is just getting referenced. It is not an internal property of a Rocket object. So changes of the prototype will therefore apply to all objects that reference it, shown in this example:

```
1 Rocket.prototype.destination = [];
2 
3 rocket1.destination.push("Orbit");
4 rocket2.destination.push("Mars");
5 
6 console.log(rocket1.destination);         //["Orbit", "Mars"]
7 console.log(rocket2.destination);         //["Orbit", "Mars"]
```

Example 10: `rocket1` and `rocket2` affect the same array

① The `destination` property is defined on the prototype which means that `rocket1.destination` and `rocket2.destination` point to the same array of values. Adding values will therefore affect both `rocket1` and `rocket2`. While this can be useful at times, it might not be the effect we are looking for. So it is very important to be careful what to define on the prototype. Note that destination is created after the creation of `rocket1` and `rocket2`. That shows again that JavaScript is very flexible and the experience is seamless.

## 3.2 Useful methods

In this section we present some useful methods when working with prototypes.

With the `hasOwnProperty()` method you can check if a property is stored internally on an object. In this example the `startEngines()` method is stored on the prototype.

```
console.log(rocket1.hasOwnProperty(startEngines()));          //false
console.log(rocket2.hasOwnProperty(startEngines()));          //false
console.log(Rocket.prototype.hasOwnProperty(startEngines())); //true
```

Example 11: Testing of `hasOwnProperty()` method

The `isPrototypeOf()` method determines whether an object is the prototype of another.

```
console.log(Rocket.prototype.isPrototypeOf(rocket1));          //true
console.log(Rocket.prototype.isPrototypeOf(rocket2));          //true
```

Example 12: Testing of `isPrototypeOf()` method

## 3.3 Prototype chains

As mentioned before inheritance does occur between objects themselves without the need of a class like structure. The model which is used in JavaScript is called **prototype chaining** or **prototypal inheritance**. Objects of the same object-type inherit throught the `[[Prototype]]` attribute. Because the prototype is also an object, it has its own prototype to inherit from which itself has another prototype and so on. That is called the **prototype chain**.

All objects in JavaScript automatically inherit from Object, meaning that every objects last prototype is `Object.prototype`, except if you specify otherwise. Methods mentioned before like `hasOwnProperty()` or `isPrototypeOf()` are defined on Object.prototype and therefore inherited and usable by almost all existing objects. The following figure shows the prototype chain for our Rocket objects.

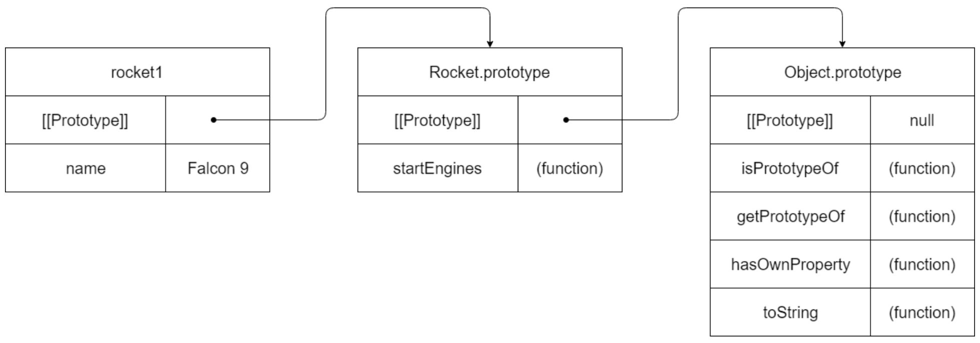

Figure 2: `Prototype chain` including `rocket1`, `Rocket.prototoype`, and `Object.prototype`

The `[[Prototype]]` attribute from rocket1 points to `Rocket.prototype`, which has a `[[Prototype]]` attribute pointing to `Object.prototype`. `Object.prototype` defines a bunch of methods which are inherited by Rocket objects, meaning they can be accessed from `rocket1`. `Object.prototype` defines some more default methods which we didn't list. It is possible to edit `Object.prototype` and add functions to it, but being pretty universal it will sooner or later cause some problems in the system. It is not advised to change it other than to experiment with it to understand **prototypal inheritance**.

## 3.4 Overriding

It is possible in JavaScript to have functions with the same name on different levels in the prototype chain. The engine looks first for an own property with that name, after that it runs up the prototype chain. That means that functions lower in the chain override those above. In the following example we replace the `toString()` method defined on `Object.prototype`.

```
1 rocket1.toString();                          //"[object Object]"
2
3 rocket1.toString = function(){
4     return "Rocket: " + this.name;
5 }
6
7 rocket1.toString();                          //"Rocket: Falcon 9"
8 delete rocket1.toString;
9 rocket1.toString();                          //"[object Object]"
```

Example 13: Overriding `toString()` Method

The `toString()` method comes from `Object.prototype` and returns `"[object Object]"` by default. ③ By replacing it with a `toString()` method on `rocket1` lower in the chain the output changes. ⑦ Because this new method is an own property it is used instead of the one in the prototype. The own property shadows the prototype property. ⑧ Only by deleting the property from the object the prototype property is accessible again. To further understand the mechanics the code is visualized in the next figure.

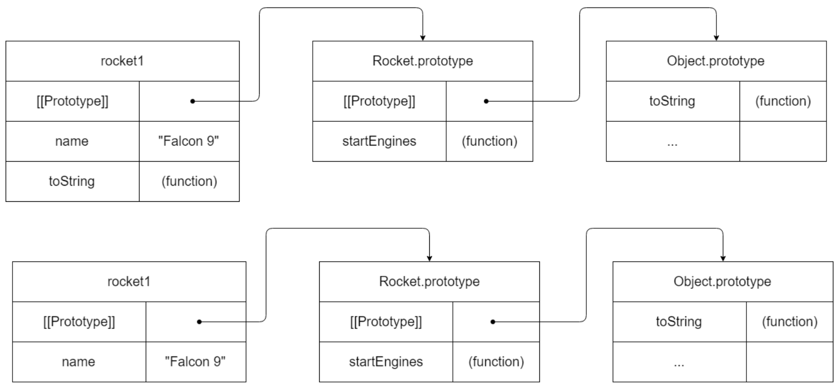

Figure 3: The `toString()` method defined on `rocket1` is shadowing `toString()` on `Object.prototype` until it is deleted

### 3.5 Object Inheritance

JavaScript makes it very simple to build a line of inheritance between objects. The only thing we must do is to set the `[[Prototype]]` attribute to the object it should inherit from. Every object has its `[[Prototype]]` set to `[ConstructorName].prototype` implicitly, but we can also explicitly choose the prototype with the `Object.create()` method. `Object.create()` needs two arguments. First an object that will be prototype and then a list of object properties. For example.

```
 1 rocket1.reusable = true;
 2
 3 var rocket3 = Object.create(rocket1, {
 4     name:{
 5         value: "Falcon 1",
 6         configurable: true,
 7         enumerable: true,
 8         writable: true
 9     }
10 });
11
12 rocket3.name;                                  //"Falcon 1"
13 rocket1.reusable;                              //true
14 rocket2.reusable;                              //undefined
15 rocket3.reusable;                              //true
16 rocket3.startEngines();                        //"Starting engines"
17 console.log(rocket1.isPrototypeOf(rocket3));   //true
```

Example 14: Creating `rocket3` with the `Object.create()` method

① This code adds a property `reusable` to `rocket1` and ③ creates a third rocket with `Object.create()`. The prototype of `rocket3` is defined to `rocket1`. ⑤ `rocket3` gets a `name` property which will shadow the `name` property of `rocket1`. ⑫ The output of `rocket3.name` is `"Falcon 1"` as expected. If there wouldn't be a name property on `rocket3` the output would be `"Falcon 9"`. ⑭ The `reusable` attribute is invisible for `rocket2` therefore `"undefined"` is the output. We know that the real Falcon Heavy is reusable but for the sake of argument it doesn't have that property.

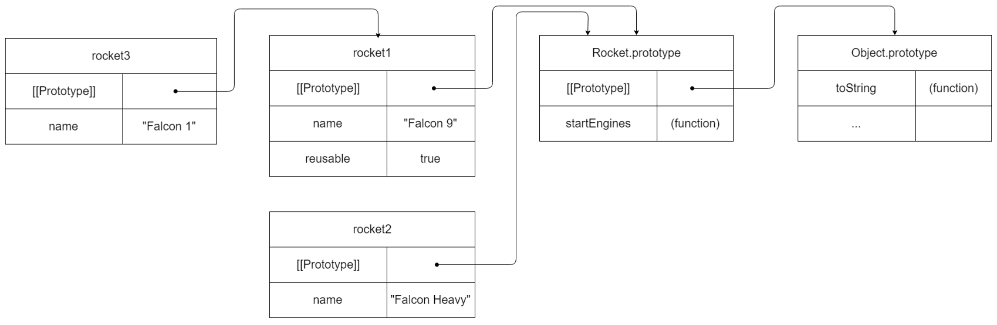

Figure 4: `Prototype chain` of rocket objects

The **prototype chain** of `rocket3` is different to `rocket2` because the prototype of `rocket3` is explicitly set to `rocket1`. The method `startEngines()` is accessible on `rocket3` as well because it is defined in its **prototype chain**. Keep in mind that although the property `reusable` is defined on an object of Rocket, it's still not available for `rocket2` because `rocket1` and `rocket2` don't know from each other.

The prototype chain is essentially the equivalent to the class structure in Java. In Java you have a class structure with mother classes and subclasses which build an inheritance system. With these classes you can create objects that have access to every method and property defined in the constructor and all the public methods and properties of the mother classes.

In JavaScript however, objects themselves are linked together with other objects. Through the `[[Prototype]]` attribute every object can see its prototype object higher up in the chain. With this prototype chain it is possible to replicate the classlike structure and achieve the same effect.

With both systems you can effectively create objects of a specific type that are capable of using methods and accessing properties that aren't defined in their class (Java) or on the object (JavaScript).

## 4 Summary

In our paper we discussed how objects and functions work. Functions act like objects and you can add and remove properties even after an object is created. The `Object.freeze()` method puts objects in a state where they are non configurable, non writable, just read-only. Constructors define default properties for objects.

A prototype is an object that is getting referenced by the `[[Prototype]]` attribute from an object. It defines properties for objects created with a particular constructor. You can add and remove properties to the prototype just as well as normal objects.

When accessing a property on an object, JavaScript first looks for an own property and then one in the prototype chain. When a prototype changes it will change the properties available for the objects that reference it as well. You can assign custom prototypes to objects with the `Object.create()` method. By that you can build your model of inheritance as you need it.

## 5 References

This paper is written closely with the book `The Principles of Object-Oriented JavaScript` from `Nicholas C. Zakas` [1]. Purpose of this paper is to shorten and simplify the topic to give a quick overview.

[1] Nicholas C. Zakas. The Principles of Object-Oriented JavaScript. No Starch Press, inc., San Francisco, 2014. ISBN 978-1-59327-540-2