JavaScript Technology

2015
Pages 1–3

Object-oriented JavaScript

Adrian Thilo, Gabriel Brandersky

1 INTRODUCTION

The aim of this paper is to explain object-oriented (OO) features of JavaScript (JS). It is expected that the reader is already familiar with basic language constructs – variables, conditionals, loops, etc. But prior knowledge of the object-oriented programming (OOP) in other languages is not necessary. In OOP, the program can be seen as a collection of objects which communicate by sending messages to solve a problem, not just as a list of functions. In the following sections, all fundamental concepts used in OOP will be described one by one: abstraction, encapsulation, polymorphism and inheritance. Each individual concept will be accompanied by at least one specific example.

2 ABSTRACTION

Abstraction is a concept used to reduce complexity and make problems easier to cope with. The real world contains far too much information which are not needed to implement a program. Therefore, abstraction is the concept of focusing on the important aspects of a problem using only relevant information. There are different levels of abstraction; at each of them, the complexity and the details shown are reduced. In OOP, abstraction is used to model real-world entities which are represented as objects in programming languages. These objects store attributes as well as important behaviour, which is represented by methods.

3 ENCAPSULATION

The concept of encapsulation is to bundle data and functions into a self-contained component. An important part of encapsulation is information hiding. This means that a component works as a black box. It hides all information inside and is connected to the outside only by interfaces. Other components are able to use just public interface parts. This interface often makes use of the private parts which can not be used directly. Implementation of a component remains hidden inside, so encapsulation provides the possibility to change internal details, e.g. change of data structure. Keywords such as private, protected, package or public are often accompanied to variables in some OO languages. This sets the visibility and shows how the variables should be used. However, JavaScript does not support encapsulation and information hiding directly. But there are some possibilities to use this concept anyway. One of them is closures. Unlike in most other object oriented languages, JavaScript does not delete variables stored in an outer function after this function is finished if these variables are still referred to in an inner function. They can not be accessed from the outside, but can still be used by the inner function. This is shown in the following example, where "`pascal`" is a function, which returns an object saved in "`language`". The variable "`freeUses`" and the function "`support`" are defined in the "`pascal`" function, but not on the object returned, which makes them private. The returned object still has access to them. The function "`use`" of this object invokes "`support`" and outputs "`ok`" or "`not ok`" depending on the current value of "`freeUses`". The variable "`freeUses`" can not be accessed on the language object, so the "`hasOwnProperty`" will result in "`false`".

```
var pascal = function() {
  var freeUses = 10;
  var support = function() {
    return (freeUses >= 0) ? 'ok' : 'not ok';
  };
  return {
    use: function() {
      freeUses -= 1;
      return support();
    }
  };
};
var language = pascal();
console.log(language.use()); // 'ok'
console.log(
  language.hasOwnProperty('freeUses')
); // false
```

Another way to achieve encapsulation in JavaScript is use of getter and setter methods. They provide an interface for using and modifying properties of objects without breaking encapsulation. The internal information is not accessed directly. If a getter and a setter are defined for the same property, then it can be used like property and not as methods. Furthermore, getters and setters provide the possibility of doing additional work. The following example shows the getter and the setter method for the property "`fullName`", which splits into "`name`" and "`version`". "`fullName`" can now be output and modified like a property.

```
var js = {
  name: 'JavaScript',
  version: 'ECMAScript 5',
  get fullName() {
    return this.name +
      ' (' + this.version + ')';
  },
  set fullName(newName) {
    var start = newName.indexOf('(');
    var end = newName.indexOf(')');
    this.name = newName.substring(0, start);
    this.version =
      newName.substring(start+1, end);
  }
}
console.log(js.fullName);
  // 'JavaScript (ECMAScript 5)'
js.fullName = 'JavaScript (ECMAScript 6)';
console.log(js.version); // 'ECMAScript 6'
```

4 POLYMORPHISM

The polymorphism is another quite simple, but powerful OO concept. The word polymorphism is derived from two words poly and morph. Poly means multiple and morphism is a state of having form. These two parts together mean having multiple forms. Polymorphism defines an interface that is shared across multiple objects, while each object provides implementation on its own. The advantage of polymorhic interface is that you can treat various objects equally without having to concern yourself about which object it really is. In the following example, the purpose is to be able print constructs in various programming languages. At first, there is a typical imperative implementation, which will then be transformed to OO code with use of polymorphism. The function "`languageVariable`" takes three arguments which are strings containing language, variable name and initial value, it returns language-specific syntax for variable definition. There is a switch statement inside and depending on the value of a language variable corresponding block is executed and the concatenated string is returned.

```
function languageVariable(lang, name, value) {
  switch(lang) {
    case 'JavaScript':
      return 'var '+ name + ' = '
        + value + ';';
    case 'Ruby':
      return name + ' = ' + value;
    // ...
  }
}
```

In OOP implementation, an object is created for every programming language, e.g. JavaScript or Ruby. Then a method for printing a variable definition is implemented for each language (as in the corresponding case statement). In order to take advantage of polymorphism, this method would have the same name in all programming languages. This allows us to eradicate the need for a switch statement. Therefore, this function would be simplified to one polymorphic method call. On the first line in the OO version the object which provides this polymorhic interface is created. How exactly the object is created will be discussed in the following section. It is important that a different programming language object could have been created, which would change the output accordingly. It is the job of a programmer to make sure that, when you call some method on an object, that it has been defined. The real advantage of the OO version is that it makes code maintainable and less coupled. Possible changes in an object-oriented version are localized to a single object. If you would like to add a new language, then you just create a new object and define this polymorphic interface. On the other hand, in the imperative version, changes would have been scattered all over the program.

```
var lang = new JavaScript(); // or new Ruby()
lang.block(function() {
  return language.variable('answer', 42);
});
```

5 INHERITANCE

Inheritance is a heavily used mechanism of code reuse. It is very common that some functionality is required across multiple types of objects. In order to prevent duplication, general objects are defined and then extended to create specialized objects which possess the functionality of general objects. A general object is referred as a parent, and a specialized object is a child. Then it is said that a child inherits all the functionality of a parent. This enables us to build a full inheritance hierarchy and capture static and dynamic relationships among objects. Similarly in JS, the attributes and methods of objects can be shared. Because JS is an OO language without the formal syntax, there exist multiple ways to implement OO concepts. To provide you with an example of inheritance, these OO patterns will be shown: prototypal, functional and pseudoclassical. Although the following examples will concentrate on inheritance, these patterns are not limited to it. For example, the second functional pattern is generally known as a factory design pattern and provides encapsulation as well. On the first line in the next code snippet, a first object is created with two properties. Then it is cloned in order to create the second object. At this moment, the second object does not contain its own properties and one property will be added on the third line. In prototypal inheritance, objects are created directly from other objects whose properties they fully inherit. Only differences from its parent are specified. Therefore, the parent of the second object is the first object, the parent of the first object was implicitly set to the object "`Object`", which has nil prototype. This is where the so called prototype chain ends. When a property is accessed on an object, but it is not found on the object itself, then the search will continue up the prototype chain until it is found or the end of the prototype is reached. But setting values never traverses the prototype chain. If the value does not exist, it will be defined; otherwise an existing value is changed. Similarly, both properties of the first object are modified, and then the same properties are accessed on the second object. Since it possesses its own "a" property, the value is immediately returned. But when property "b" is accessed, which is not present on the second object, the search will continue on its parent, the first object, where it will be found. Notice that the modified value has been returned and not the value present at the moment of creation. This is the difference between cloning and copying. When a property is just copied, the changes to an original property are not reflected in its copies. In case that the end of the prototype chain is reached then "undefined" is returned.

```
var obj1 = {a: 'obj1_a', b: 'obj1_b'};
var obj2 = Object.create(obj1);
obj2.a = 'obj2_a';
console.log(Object.getPrototypeOf(obj1));
  // Object {...}
console.log(Object.getPrototypeOf(obj2));
  // obj1 {...}
obj1.a += '_modified';
obj1.b += '_modified';
console.log(obj2.a); // obj2_a
console.log(obj2.b); // obj1_b_modified
console.log(obj2.c); // undefined
```

As described by Douglas Crockford, the functional pattern of inheritance uses simple functions (not constructor functions described below). Therefore, the functional pattern can be used in conjunction with functional features of JavaScript, e.g. "bind" method. This pattern is especially helpful when the process of creating is more complicated and includes multiple steps because it can be tedious and error generating to specify all differences. Another advantage is encapsulation meaning that private variables declared to be inside function are not accessible from outside. This pattern can be combined with other inheritance patterns, which is why it is the most expressive pattern. This pattern is just mentioned for the sake of completness, but without an example. The last pseudoclassical pattern was designed to appear similar for programmers who are already familiar with classical languages. But under the hood, it still uses prototypal inheritance, which is not going to change any time soon. Even though there is a class keyword in ECMAScript 6, it is once again just syntax sugar on top of existing features. Every function can act as a constructor function, which is often called class, and objects created from it are its instances. Every constructor is accompanied by a special empty object "prototype". The convention is to use a first uppercase letter to signify that a new operator must be used, as in this example of "Language" and "Ruby". When a class is instanciated, a new empty object is created. Then the object's special properties are set: "constructor" and "prototype", which will be equivalent to properties of a class. Finally, a constructor function is called with the context of a new object available as "this" variable. It can be used to define properties and methods present on every instance of this class. Properties shared across all instances can be defined by class prototype because every instance points to it. So "name" property is available on every instance, but "getName" is defined by prototype object because it should not be duplicated on every instance. In order to subclass a class, we overwrite the prototype object created by default with a new instance of a superclass "Language". Otherwise, everything remains the same for subclass "Ruby" and it is possible to define other methods.

```
var Language = function(name) {
  this.name = name;
};
Language.prototype.getName = function() {
  return this.name;
};
var Ruby = function() {};
Ruby.prototype = new Language('Ruby');
Ruby.prototype.var = function(name, value) {
  return name + ' = ' + value;
}
var ruby = new Ruby();
console.log(ruby.getName()); // "Ruby"
```

6 CONCLUSION

This paper showed the concepts of OOP and how they are realized in JavaScript, which supports prototypal OOP. Abstraction for managing complexity is intuitively used to model problems. While encapsulation is not directly supported by JavaScript, there are some features like closures or getter and setter methods to achieve encapsulation and data hiding. Polymorphism is well supported by JavaScript, as different objects provide a single interface, as long as the properties have the same name. For inheritance there are different patterns to create new objects. These patterns have different advantages and disadvantages. The basic prototype pattern is simple to use. The functional pattern offers support for creating an object in more steps and provides encapsulation. Pseudoclassical appears like classical OOP, but uses prototype architecture. These patterns can also be combined. All in all, it can be seen that JavaScript offers more than imperative programming, as it is full OO language. Therefore, all concepts mentioned before can be used to improve the quality of JavaScript programs and achieve the advantages of OOP such as better code reuse and maintenance.

REFERENCES

Crockford, D. *JavaScript: The Good Parts*, 1st edn. O'Reilly Media, Beijing, 2008.

Franklin, J., *JavaScript Getters and Setters*, 2013.

Mozilla Developer Network, *Introduction to Object-Oriented JavaScript*, 2015. [online] [Accessed 3 Sep. 2015]. Available at:

```
https://developer.mozilla.org/en-US/docs/Web/JavaScript/
  Introduction_to_Object-Oriented_JavaScript
```