Menu

ECMA-262

by Dmitry Soshnikov

ECMA-262-3 in detail. Chapter 7.1. OOP: The general theory.

Tweet Like 12 G+1 7 Share 1 Share

Read this article in: Russian.

Introduction

In this article we consider major aspects of object-oriented programming in ECMAScript. That the article has not turned to “yet another” (as this topic already discussed in many articles), more attention will be given, besides, to theoretical aspects to see these processes from within. In particular, we will consider algorithms of objects creation, see how relationships between objects (including the basic relationship — inheritance) are made, and also give accurate definitions which can be used in discussions (that I hope will dispel some terminological and ideological doubts and messes arising often in articles on OOP in

JavaScript).

General provisions, paradigms and ideology

Before analysis of technical part of OOP in ECMAScript, it is necessary to specify a number of general characteristics, and also to clarify the key concepts of the general theory.

ECMAScript supports multiple programming paradigms, which are: structured, object-oriented, functional, imperative and, in the certain cases, aspect-oriented; but, as article is devoted to OOP, let us give the definition of ECMAScript concerning this essence:

ECMAScript is the *object-oriented* programming language with the *prototype based* implementation.

Prototype based model of OOP has a number of differences from the *static class based* paradigm. Let's take a look at them in detail.

Features of class based and prototype based models

Notice, in the previous sentence an important point has been noted — exactly *static* class based. With a "static" term we understand static objects and classes and, as a rule, strong typing (though, the last is not required).

This case is noticed, because often enough in various articles and on forums, calling JavaScript *"another"*, *"different"* the main reason can be used as an oppose pair: *"class vs. prototype"*, although only this difference in some implementations (e.g. in dynamic class-based Python or Ruby) is not so essential (and accepting some conditions, JavaScript becomes not so "another", though the differences in the certain ideological features, nevertheless, are). But more essential is the opposition of *"statics + classes vs. dynamics + prototypes"*. Exactly a statics and classes (examples: C++, Java) and related mechanisms of properties/methods resolution allow to see an accurate difference from the prototype based implementation.

But let us give one after another. Let's consider the general theory and key concepts of these paradigms.

Static class based model

In the class based model there is a concept of a *class* and an *instance* which belongs to this classification. Instances of a class are also often named as *objects* or *exemplars*.

Classes and objects

The class represents a *formal abstract set* of the generalized characteristics of an instance (the *knowledge* about objects).

The term *set* in this respect is closer to mathematics, however, it is possible to call it as *type* or *classification*.

Example (here and below examples will be given in pseudo-code):

```
C = Class {a, b, c} // class C, with characteristics a, b, c
```

Characteristics of instances are: *properties (object description)* and *methods (object activity)*.

Characteristics themselves also can be treated as objects: i.e. whether a property is writable, configurable, active (getter/setter), etc.

Thus, objects store a *state* (i.e. concrete values of all properties described in a class), and classes define *strict unchangeable structure* (i.e. presence of those or other properties) and *strict unchangeable behavior* (presence of those or other methods) of their instances.

```
C = Class {a, b, c, method1, method2}

c1 = {a: 10, b: 20, c: 30} // object c1 of the class C
c2 = {a: 50, b: 60, c: 70} // object c2 with its own state, of the same
```

Hierarchical inheritance

For improvement of *a code reuse*, classes can *extend* other classes, bringing necessary additions. This mechanism is called as *(hierarchical) inheritance*.

```
D = Class extends C = {d, e} // {a, b, c, d, e}
d1 = {a: 10, b: 20, c: 30, d: 40, e: 50}
```

The resolution of methods when calling methods from instances is handled by *strict, invariant and consecutive* examination of the class on presence of this or that method; if the method is not found in the native class, search in the class-ancestor, in the ancestor of the class-ancestor etc. i.e. in the *strict hierarchical chain* proceeds. If on reaching the base link of inheritance chain the method still remains unresolved, the conclusion is: *the object does not have (in its set and its hierarchical chain) similar behavior, and to get desirable result is not possible*.

```
d1.method1() // D.method1 (no) -> C.method1 (yes)
d1.method5() // D.method5 (no) -> C.method5 (no) -> no result
```

In contrast with methods which at inheritance are not copied into a class-descendant but form hierarchy, properties are always copied. We can see this behavior on the example of class `D` which ancestor is the class `C`: properties `a`, `b` and `c` are copied, and the structure of D is: `{a, b, c, d, e}`. However, methods `{method1, method2}` are not copied, but inherited. Therefore, the memory usage in this aspect is directly proportional to the depth of hierarchy. The basic lack here is that even if at deeper levels of a hierarchy some properties are not needed to object, it will have all of them anyway.

Key concepts of class based model

So, we have the following key concepts:

- to create an object first it is necessary to define its class;
- thus, the object will be created in its own classification "image and similarity" (structure and behavior);
- resolution of methods is handled by a strict, direct, unchangeable chain of *inheritance*;
- classes-descendants (and accordingly, objects created from them) contain all properties of an inheritance chain (even if some of these properties are not necessary to the concrete inherited class);

- being created, the class cannot (because of the static model) to change a set of characteristics (neither properties, nor methods) of their instances;
- instances (again because of strict static model) cannot have neither additional own (unique) behavior, nor the additional properties which are distinct from structure and behavior of the class.

Let’s take a look at what the alternative OOP model, based on prototypes, suggests.

Prototype based model

Here the basic concept is *dynamic mutable objects*.

Mutations (full convertibility: not only values, but also all of characteristics) are directly related with dynamics of the language.

Such objects can independently store all their characteristics (properties, methods) and do not need the class.

```
object = {a: 10, b: 20, c: 30, method: fn};
object.a; // 10
object.c; // 30
object.method();
```

Moreover, because of dynamics, they can easily change (add, delete, modify) their characteristics:

```
object.method5 = function () {...}; // add new method
object.d = 40; // add new property "d"
delete object.c; // remove property "c"
object.a = 100; // modify property "a"

// as a result: object: {a: 100, b: 20, d: 40, method: fn, method5: fn}
```

That is, at assignment if some characteristic does not exist in object, it is created and initialized with passed value; if it exists, it is just updated.

The code reuse in this case is achieved not via extending the classes (note, we don’t say a word about any classes as sets of unchangeable characteristics; the classes are not present here at all), but by referencing to, so-called, *prototype*.

Prototype is an object, which is used either as an *original copy* for other objects, or as a *helper object to which characteristics other objects can delegate* in case if these objects do not have the necessary characteristic themselves.

Delegation based model

Any object can be used as a prototype of another object and again because of mutations the object can easily change its prototype dynamically at runtime.

Notice, currently we’re considering the general theory, not touching concrete implementations; when we will discuss concrete implementations (and in particular, ECMAScript), we will see a number of own features.

Example (pseudo-code):

```
x = {a: 10, b: 20};
y = {a: 40, c: 50};
y.[[Prototype]] = x; // x is the prototype of y

y.a; // 40, own characteristic
y.c; // 50, also own
y.b; // 20 – is gotten from the prototype: y.b (no) -> y.[[Prototype]]

delete y.a; // removed own "a"
y.a; // 10 – is gotten from the prototype

z = {a: 100, e: 50}
y.[[Prototype]] = z; // changed the prototype of the y to z
y.a; // 100 – is gotten from the prototype
y.e // 50, also – is gotten from the prototype

z.q = 200 // added new property to the prototype
y.q // changes are available and on y
```

This example shows the important feature and the mechanism related with a prototype when it is used as a helper object to which properties, in case of absence of own similar properties, other objects can delegate.

This mechanism is called a *delegation* and based on it prototypical model is a *delegating prototyping (or a delegation based prototyping)*.

Referencing to the characteristics in this case is called a *sending a message* to an object. I.e., when the object cannot respond to the message itself, it delegates to the prototype (asking it to try to answer the message).

The code reuse in this case is called *delegation based inheritance* or *prototype based inheritance*.

Since any object can be used as a prototype, it means that prototypes can also have their own prototypes. This connected combination of prototypes forms a, so-called, *prototype chain*. The chain also like in static classes is *hierarchical*, however because of mutations it can easily rearrange, changing hierarchy and the structure.

```
x = {a: 10}

y = {b: 20}
y.[[Prototype]] = x

z = {c: 30}
z.[[Prototype]] = y

z.a // 10

// z.a found in prototype chain:
// z.a (no) ->
// z.[[Prototype]].a (no) ->
// z.[[Prototype]].[[Prototype]].a (yes): 10
```

If an object and its prototype chain could not respond to the sent message, the object can activate a corresponding *system signal*, handling which it is possible to continue dispatching and delegation to another chain.

This system signal is available in many implementations, including the dynamic class based systems: it's *#doesNotUnderstand* in SmallTalk; *method_missing* in Ruby; *__getattr__* in Python; *__call* in PHP; *__noSuchMethod__* in one of ECMAScript implementations, etc.

Example (SpiderMonkey ECMAScript implementation):

```
var object = {

  // catch the system signal about
  // impossibility to respond the message
  __noSuchMethod__: function (name, args) {
    alert([name, args]);
    if (name == 'test') {
      return '.test() method is handled';
    }
    return delegate[name].apply(this, args);
  }

};

var delegate = {
  square: function (a) {
    return a * a;
  }
};

alert(object.square(10)); // 100
alert(object.test()); // .test() method is handled
```

That is, in contrast with the static class based implementation, in case of impossibility to respond the message, the conclusion is: *the object at the moment does not have the requested characteristic, however to get the result is still possible if to try to analyze alternative prototype chain, or probably, the object will have such characteristic after a number of mutations*.

Regarding ECMAScript, here exactly this implementation — *delegation based prototyping* is used. However, as we will see, up to the specification and implementations there are also some own features.

Concatenative model

But for to be fair, it is necessary to tell some words about other case from definition (though it is not used in ECMAScript) when prototype represents original object from which other object are copied.

Here for a code reuse is used not a delegation but *exact copy (a clone)* of a prototype at the moment of object's *creation*.

This kind of prototyping is called as *concatenative* prototyping.

Having copied in itself all of prototype's characteristics, an object can further fully change its properties and methods also as a prototype can change its own (and this changes will not affect on already existing objects as it would be with changing prototype's characteristics in delegation based model). As the advantage of such approach can be decreasing of the time for dispatching and delegation and the basic lack is higher memory usage.

"Duck" typing

Coming back to dynamics, weak typing and mutations of objects, in contrast with the static class based model, passage of the test for ability to do some actions here can be related *not with what type (class) the object has*, but whether *it is able to respond the message* (whether it is capable to do that will be required after passing the test).

Example:

```
// in static class based model
if (object instanceof SomeClass) {
  // some actions are allowed
}

// in dynamic implementation
// it is not essential what the type of an object
// at the moment, because of mutations, type and
// characteristics can be transformed
// repeatedly, but essential, whether the object
// can respond the "test" message

if (isFunction(object.test)) // ECMAScript

if object.respond_to?(:test) // Ruby

if hasattr(object, 'test'): // Python
```

On jargon, it's called as duck typing. That is, objects can be identified by set of their characteristics which are present at the moment of check, instead of object's position in hierarchy or their belonging to any concrete type.

Key concepts of prototype based model

So, let's check the main features of this approach:

- the basic concept is an *object*;
- objects are fully dynamic and mutable (and in the theory can completely mutate from one type into another);
- objects do not have the strict classes which describe their structure and behavior; objects do not need classes;
- however, not having classes, objects can have prototypes to which they can delegate if cannot answer the message themselves;
- the object prototype can be changed at any moment at runtime;
- in *delegation based* model changing prototype's characteristics will affect on all objects related with this prototype;

- in *concatenative prototype* model prototype is the *original copy* from which other objects are cloned and further become completely independent; changes of prototype's characteristics do not affect on objects cloned from it;
- if it is not possible to respond to a message, it is possible to signal the caller about it which can take additional measures (for example, to change dispatching);
- identification of objects can be made not by their hierarchy and belonging to concrete type, but by a current set of characteristics.

However, there is one more model which we should consider also.

Dynamic class based model

We consider this model to show on the example what has been mentioned in the beginning — the difference between just *"a class vs. a prototype"* is not so essential (*especially if the prototype chain is invariant*; for more accurate distinction, it is necessary to consider also a *statics* in classes). As an example, it is possible to use Python or Ruby (or other similar languages). Both languages are use dynamic class based paradigm. However, in certain aspects, some features of prototype based implementation can be seen.

In the following example we can see that just like in the delegation based prototyping, we can augment a class (prototype), and it affects all objects related with this class, we can also dynamically, at runtime, to change object's class (providing a new object for delegation) etc.

```
# Python

class A(object):

    def __init__(self, a):
        self.a = a

    def square(self):
        return self.a * self.a

a = A(10) # creating an instance
print(a.a) # 10

A.b = 20 # new property of the class
print(a.b) # 20 – available via "delegation" for the "a" instance

a.b = 30 # own property created
print(a.b) # 30

del a.b # removed own property
print(a.b) # 20 - again is taken from the class (prototype)

# just like in prototype based model
# it is possible to change "prototype"
# of an object at runtime

class B(object): # "empty" class B
    pass

```

```
b = B() # an instance of the class B

b.__class__ = A # changing class (prototype) dynamically

b.a = 10 # create new property
print(b.square()) # 100 - method of the class A is available

# we can delete explicit references on classes
del A
del B

# but the object still have implicit
# reference and the methods are still available
print(b.square()) # 100

# but, to change the class on some of build-in
# is impossible (in current version) and this is the
# feature of implementation
b.__class__ = dict # error
```

In Ruby the picture is similar: there also fully dynamic classes are used (by the way, in current version of Python, in contrast with Ruby and ECMAScript, it is impossible to augment built-in classes/prototypes), we can completely change characteristics of objects and classes (to add methods/properties in a class, and these changes will affect on already existing objects); however, for example, it is impossible to change the class of an object dynamically.

But, this article is dedicated not to Python and Ruby; therefore we're finishing this comparison, and starting to discuss ECMAScript itself.

But before, we still need to take a look on additional *"syntactic and ideological sugar"*, available in some OOP implementations, because such questions often appear in some articles about JavaScript.

And this section has been considerate only to note incorrectness of statements like *"JavaScript is another, it has prototypes, instead of classes"*. It is necessary to understand that *not all* class based implementations are *completely different* in their implementation. And even if we may talk that *"JavaScript is different"* it is necessary to consider (besides only the concept of a "class") also all other related features.

Additional features of various OOP implementations

In this section we will take a brief look on additional features and kinds of code reuse in various OOP implementations, making a parallel with ECMAScript's OOP implementation. The reason is that in appearing articles on JavaScript, OOP concept is limited to some habitual implementation, regardless that there can be various implementations; the only (and main) requirement is that they should technologically and ideologically be proved. Without having found similarity with some "syntactic sugar" from one (habitual) OOP implementation, JavaScript can hasty be named as a "not pure OOP language" that is the incorrect statement.

Polymorphism

Objects in ECMAScript are polymorphic in several meanings.

For example, one function can be applied to different objects, just like it would be the native characteristic of an object (because this value determinate on entering the execution context):

```
function test() {
  alert([this.a, this.b]);
}

test.call({a: 10, b: 20}); // 10, 20
test.call({a: 100, b: 200}); // 100, 200

var a = 1;
var b = 2;

test(); // 1, 2
```

However, there are exceptions: for example, method *Date.prototype.getTime()*, by the standard, as *this* value always should have a date object, otherwise, the exception is being thrown:

```
alert(Date.prototype.getTime.call(new Date())); // time
alert(Date.prototype.getTime.call(new String(''))); // TypeError
```

Or so-called *parametric polymorphism* when function is defined equally for all data types, but however, accepts polymorphic functional argument (example is the *.sort* method of arrays and its argument — polymorphic sort function). By the way, the example above also can be treated as a kind of parametric polymorphism.

Or in the prototype the method can be defined empty, and all created objects should redefine (implement) this method (i.e. "one interface (signature), and a lot of implementations").

Also polymorphism here can be related with *duck typing* which we mentioned above: i.e. the type of object and its place in hierarchy are not so important, but if it has all necessary characteristics, it can be easily accepted (i.e., again the general interface is important, and implementations can vary).

Encapsulation

With this idea there is often a mess and errors in perception. In this case we discuss one of convenient "sugars" of some OOP implementations — well known modifiers: *private, protected* and *public* which also are called as *access levels (or access modifiers)* to characteristics of objects.

I want to note and remind the main purpose of the encapsulation essence: *encapsulation* is an *increasing of abstraction*, but not a paranoid hiding from "malicious hackers" which, "want to write something directly into fields of your classes".

It is a big (and widespread) *mistake* to use *hiding for the sake of hiding*.

Access levels (private, protected and public), have been provided in several OOP implementations first of all for *convenience of the programmer* (and, really, are convenient enough "sugar") to more abstractly describe and build system.

This can be seen in some implementations (e.g. in already mentioned Python and Ruby). On one hand (in Python), these are *__private* and *_protected* properties (which are specified by naming convention via leading underscores) and are not accessible from the outside. On the other hand, Python simply renames such fields by special rules *(_ClassName__field_name)*, and by this name they are already *accessible from the outside*.

```
class A(object):

    def __init__(self):
      self.public = 10
      self.__private = 20

    def get_private(self):
        return self.__private

# outside:

a = A() # instance of A

print(a.public) # OK, 30
print(a.get_private()) # OK, 20
print(a.__private) # fail, available only within A description

# but Python just renames such properties to
# _ClassName__property_name
# and by this name theses properties are
# available outside

print(a._A__private) # OK, 20
```

Or in Ruby: on one hand, there is an ability to define private and protected characteristics; on the other hand, there are special methods (e.g. *instance_variable_get*, *instance_variable_set*, *send* etc.), which allow to get access to encapsulated data.

```
class A

  def initialize
    @a = 10
  end

  def public_method
    private_method(20)
  end

private

  def private_method(b)
    return @a + b
  end

end

a = A.new # new instance

a.public_method # OK, 30

```

```
a.a # fail, @a - is private instance variable without "a" getter

# fail "private_method" is private and
# available only within A class definition

a.private_method # Error

# But, using special meta methods - we have
# access to that encapsulated data:

a.send(:private_method, 20) # OK, 30
a.instance_variable_get(:@a) # OK, 10
```

The main reason is that a programmer *himself* wants to get access to the encapsulated (please notice, I'm specifically not using term "hidden") data. And if these data will be somehow incorrectly changed or there will be any error — the full responsibility for this is completely on the programmer, but not simply a "typing error" or "someone has casually changed some field". But if such cases become frequent, we may nevertheless note a *bad programming practice and style*, since usually it's the best to "talk" with objects only via public API.

The basic purpose of the encapsulation, repeat, is an *abstraction* from the user of the *auxiliary helper data* and not a "way to secure object from hackers". Much more serious measures, rather than the "private" modifier are used for software security and safety.

Encapsulating *auxiliary* helper (local) objects, we provide possibility for further behavior changes of the public-interface with a *minimum of expenses*, localizing and predicting places of these changes. *And exactly this is the main encapsulation purpose*.

Also the important purpose of a setter method is to abstract difficult calculations. For example, the *element.innerHTML* setter — we simply *abstractly* statement — *"now html of this element is the following"* while in setter function for the *innerHTML* property there will be difficult calculations and checks. In this case the issue mostly relates to the *abstraction*, but *encapsulation* as its *increasing* also takes place.

The concept of encapsulation can be related not only to OOP. For example, it can be a simple function which *encapsulates* various calculations, making its using *abstract* (it is not so important for user to know, how e.g. function Math.round(…) is implemented; he simply calls it). It is an encapsulation and, pay attention, I don't say a word about any "private, protected and public".

ECMAScript, in the current version of specification, does not define private, protected, and private modifiers.

However, on practice it is possible to see something that is named "imitation of encapsulation in JS". Usually for this purpose surrounding context (as a rule, the constructor function itself) is used. Unfortunately, often implementing such "imitation", programmers can produce "getters/setters" for *absolutely non-abstract* entities (I repeat, incorrectly treating encapsulation itself):

```
function A() {

  var _a; // "private" a

  this.getA = function _getA() {
    return _a;
  };

  this.setA = function _setA(a) {
    _a = a;
  };

}

var a = new A();

a.setA(10);
alert(a._a); // undefined, "private"
alert(a.getA()); // 10
```

Thus, everybody understands that for each created object, the pair of methods "getA/setA" is created and what causes the memory issues directly proportional to quantity of created objects (in contrast with if methods would be defined in a prototype). Although, in first case theoretically could be optimization with joined objects.

Also in various articles about JavaScript for such methods, there's a name *"privileged methods"*. To specify, note: ECMA-262-3 does not define any "privileged methods" concept.

However, it can be normal to create methods in the constructor function, as it in ideology of the language — objects are completely mutable and can have the unique characteristics (in the constructor, by a condition, some object can get an additional method, and the other — not etc.).

Moreover, regarding JavaScript, such *"hidden" "private" vars* are not so hidden (if encapsulation is still misinterpreted as protection against "the malicious hacker" which wants to write value in a certain field directly, instead of using a setter method). In some implementations, it is possible to get access to the necessary scope chain (and accordingly to all variable objects in it), by passing a calling context to the *eval* function (it can be tested in SpiderMonkey up to version 1.7):

```
eval('_a = 100', a.getA); // or a.setA, as "_a" is in [[Scope]] of both
a.getA(); // 100
```

Or, in the implementations that allow direct access to the activation object (for example, Rhino); changing value of internal variables is possible via accessing corresponding properties of that object:

```
// Rhino
var foo = (function () {
  var x = 10; // "private"
  return function () {
    print(x);
  };
})();
foo(); // 10
foo.__parent__.x = 20;
foo(); // 20
```

Sometimes, as organizational measures (which also can be treated as a kind of encapsulation), "private" and "protected" data in JavaScript are marked by a leading underscore (but in contrast with the Python, here it is only the naming convention):

```
var _myPrivateData = 'testString';
```

Regarding the effect with surrounding execution context, it’s used very often, but for encapsulation of *really* auxiliary (helper) data, which is not directly related to the object, and that’s convenient to abstract them from the external API:

```
(function () {

  // initializing context

})();
```

Multiple inheritance

Multiple inheritance is a convenient “sugar” for code reuse improvement (if we can inherit one class why not to inherit ten at once?). However, it has a number of lacks and that’s why is not popular in implementations.

ECMAScript does not support multiple inheritance (i.e. only one object can be used as a direct prototype) although its ancestor Self programming language had such ability. But in some implementations, such as SpiderMonkey, using *__noSuchMethod__*, it is possible to manage dispatching and delegation to alternative prototype chains.

Mixins

Mixins are also a convenient way of a code reuse. Mixins have been suggested as an alternative to multiple inheritance. These are independent elements which can be *mixed* with any objects, extending their functionality (thus the object can mix several mixins). ECMA-262-3 specification does not define *“mixin”* concept, however according to mixins definition, and because ECMAScript has dynamic mutable objects, nothing prevents from mixing any object with another, simply augmenting its characteristics.

Classical example:

```
// helper for augmentation
Object.extend = function (destination, source) {
  for (property in source) if (source.hasOwnProperty(property)) {
    destination[property] = source[property];
  }
  return destination;
};

var X = {a: 10, b: 20};
var Y = {c: 30, d: 40};

Object.extend(X, Y); // mix Y into X
alert([X.a, X.b, X.c, X.d]); 10, 20, 30, 40
```

Note, I take these definitions (“mixin”, we “mix”) in quotes as was mentioned that ECMA-262-3 does not define such concept and moreover it’s not a mix but usual extending of object by new characteristics (while, for example, in Ruby where the concept of mixins is official, mixin creates a *reference* to included module (i.e. as a matter of fact — creates additional object ("prototype") for delegation), instead of simply copies all properties of the module into an object).

Traits

Traits are similar to mixins, however have a number of features (basic of which is that traits, by definition, should not have a state which can make naming conflict as it could be with mixins). Regarding ECMAScript, traits are imitated by the same principle, as mixins; the standard doesn't define "traits" concept.

Interfaces

Interfaces available in some OOP implementations are similar to mixins and traits. However, in contrast with traits and mixins, interfaces force classes to completely implement behavior of signatures of their methods.

Interfaces can be treated as completely abstract classes. However, in contrast with abstract classes, (which can implement part of methods itself and other part — to define as signatures) at single inheritance a class can implement several interfaces; for this reason, interfaces (as well as mixins) can be treated as alternative to multiple inheritance.

Standard ECMA-262-3 defines neither concepts of "interface", nor concepts of "abstract class". However, as imitation, it is possible to use augmentation of objects with objects having "empty" methods (or, throwing an exception in methods, signaling that this method should be implemented).

Object composition

Object composition is also one of techniques for a dynamic code reuse. Object composition differs from inheritance with higher flexibility and implements a delegation to dynamically mutable delegates. And this, in turn, is a basis of delegation based prototyping. Besides dynamically mutable prototype, the object can aggregate (and as a result create a *composition*, an *aggregation*) object for delegation, and further at a sending of a certain message to object, to delegate to this delegate. There can be more than one delegate and because of dynamic nature it is possible to change them at runtime.

As an example already mentioned __*noSuchMethod*__ can be, but also let's show how to use delegates explicitly:

Example:

```
var _delegate = {
  foo: function () {
    alert('_delegate.foo');
  }
};

var agregate = {

  delegate: _delegate,

  foo: function () {
    return this.delegate.foo.call(this);
  }

};

agregate.foo(); // delegate.foo

agregate.delegate = {
  foo: function () {
    alert('foo from new delegate');
  }
};

agregate.foo(); // foo from new delegate
```

This relation of objects is called as *"has-a"*, i.e. *"contains inside"* in contrast with inheritance what *"is-a"* i.e. *"is a descendant"*.

As the lack of explicit composition (along with its flexibility in contrast with inheritance) the increasing of an intermediate code can be.

AOP features

As a feature of aspect-oriented programming, *function decorators* can be. Specification ECMA-262-3 doesn't define concept of "function decorators" explicitly (in contrast with, say, Python where this term is official). However, having functional arguments function can be *decorated* and activated in some *aspect* (by applying so-called *advice*):

Example of the simplest decorator:

```
function checkDecorator(originalFunction) {
  return function () {
    if (fooBar != 'test') {
      alert('wrong parameter');
      return false;
    }
    return originalFunction();
  };
}

function test() {
  alert('test function');
}

var testWithCheck = checkDecorator(test);
var fooBar = false;

test(); // 'test function'
testWithCheck(); // 'wrong parameter'

fooBar = 'test';
```

```
test(); // 'test function'
testWithCheck(); // 'test function'
```

Conclusion

At this point we're finishing consideration of the general theory (I hope this material has appeared useful to you), and continue with the ECMA-262-3 in detail. Chapter 7.2. OOP: ECMAScript implementation.

Additional literature

- Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems (by Henry Lieberman);
- Prototype-based programming;
- Class;
- Object-oriented programming;
- Abstraction;
- Encapsulation;
- Polymorphism;
- Inheritance;
- Multiple inheritance;
- Mixin;
- Trait;
- Interface;
- Abstract class;
- Object composition;
- Aspect-oriented programming;
- Dynamic programming language.

Translated by: Dmitry A. Soshnikov, with help of Juriy "kangax" Zaytsev.
Published on: 2010-03-04

Originally written by: Dmitry A. Soshnikov [ru, read »]
Originally published on: 2009-09-12 [ru]

Tweet Like 12 G+1 7 Share 1 Share

Tags: ECMA-262-3, ECMAScript, Object-oriented programming, OOP, Prototype

This entry was posted on March 04th, 2010 and is filed under ECMAScript. You can follow any responses to this entry through the RSS 2.0 feed. You can leave a response or Trackback from your own site.

« ECMA-262-3 in detail. Chapter 6. Closures.
ECMA-262-3 in detail. Chapter 7.2. OOP: ECMAScript implementation. »

6 Comments:

1. ju350213 [#permalink]
   20. November 2011 at 07:58

   > properties “d” and “e” have been copied (as a result, structure of D is: {a, b, c, d, e}); however, methods {method1, method2}) have not been copied.

   I think you have a typo here,should the sentence be *properties “a”,”b” and “c” have been copied* ??

2. Dmitry A. Soshnikov [#permalink]
   20. November 2011 at 15:49

   @ju350213

   Yep, true; thanks, fixed.

   I’ve also rewritten the sentence in the dynamic class-based model section.

3. djjj [#permalink]
   21. June 2012 at 06:16

   Hey, thanks a lot for the in-depth article – really learnt a lot from reading it. i had to read parts of it more than once (sometimes more!) to really understand the concepts but it was worth taking the time. thanks again 🙂

4. allbutone [#permalink]
   4. June 2013 at 00:25

   Assume there is a prototype Object –> parentObj;
   if i want to make a childObj based on parentObj
   the pseudo-code below is what i think would make it:

```
//in Delegation mode, inheritance can be described as:
childObj.[[prototype]] = parentObj;//establish a reference relatic
//make some changes on childObj later...
//if parentObj is changed afterwards, childObj will be affected.
//because all the properties within parentObj were referenced by c

```

```
//in Concatenative mode, inheritance can be described as:
childObj = makeCopyOf(parentObj);
//make some changes on childObj later...
//if parentObj is changed afterwards, there wouldn't be any affec
//because childObj and parentObj are two independent Objects.
```

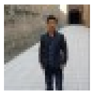
Am I correct ?

#permalink

5. Dmitry Soshnikov
5. June 2013 at 16:28

@**allbutone**, yes, this is correct.

#permalink

6.  Hong
25. July 2014 at 05:28

@**Dmitry**, thanks your excellent article.

> Features of class based and prototype based models
>
> Notice, in the previous sentence an important point has been noted — exactly static class based. With a "static" term we understand static objects and classes and, as a rule, strong typing (though, the last is not required).
>
> This case is noticed, because often enough in various articles and on forums, calling JavaScript "another", "different" the main reason can be used as an oppose pair: "class vs. prototype", although only this difference in some implementations (e.g. in dynamic class-based Python or Ruby) is not so essential (and accepting some conditions, JavaScript becomes not so "another", though the differences in the certain ideological features, nevertheless, are). But more essential is the opposition of "statics + classes vs. dynamics + prototypes". Exactly a statics and classes (examples: C++, Java) and related mechanisms of properties/methods resolution allow to see an accurate difference from the prototype based implementation.
>
> But let us give one after another. Let's consider the general theory and key concepts of these paradigms.

For above parts, I can catch your meaning, while feeling they are not as clear as other parts, especially the second paragraph `This case is noticed, ... from the prototype based implementation.`.

Just one suggestion :).

Regards.

Leave a Reply

Name (required)

Mail (will not be published) (required)

Website

Code: For code you can use tags [js], [text], [ruby] and other.

XHTML: You can use these tags: <a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite> <code> <del datetime=""> <em> <i> <q cite=""> <s> <strike> <strong>

Submit

Search...

Articles

- x86: More code – less code
- Notes. ECMAScript: Unresolved references
- OO Relationships
- x86: Generated code optimizations and tricks
- Заметки ES6: значения параметров по умолчанию

Comments

- Dmitry Soshnikov on ECMA-262-3 in detail. Chapter 7.2. OOP: ECMAScript implementation.
  @Lorenzo there is maybe a typo? you were showing that if you change the prototype the reference is lost, so...
- Lorenzo on ECMA-262-3 in detail. Chapter 7.2. OOP: ECMAScript implementation.
  Hi Dmitry, Thanks for your posts, really interesting and useful. In chapter 3.1 'Property constructor', in the following code: [js]...
- Rafal Bartoszek on The quiz
  Nice quiz, #9 Like many people I like it the most :) #6 Just for my sense of completion...

Tags

Accessor property Activation object by reference by sharing by value Closure CoffeeScript Data property ECMA-262-3 ECMA-262-5 ECMAScript ECMAScript 5 ES6 Essentials of interpretation Evaluation strategy execution context First-class objects Funarg Functional programming Function Declaration Function Expression Interpreter JavaScript lexical environment name binding Notes Object-oriented programming OOP Property Property Descriptor Property Identifier Prototype Russian Scope chain SICP this Variable object [[Scope]] Замыкание ООП Объектно-ориентированное программирование Объект переменных Прототип Фунарг контекст исполнения

Archive

- February 2016
- January 2016
- September 2015
- September 2014
- August 2014
- November 2011
- August 2011
- July 2011
- February 2011
- January 2011
- December 2010
- September 2010
- June 2010
- April 2010
- March 2010
- February 2010
- November 2009
- September 2009
- July 2009
- June 2009