Menu

ECMA-262

by Dmitry Soshnikov

ECMA-262-3 in detail. Chapter 7.2. OOP: ECMAScript implementation.

Read this article in: Russian.

Introduction

This is the second part of article about object-oriented programming in ECMAScript. In the first part we discussed the general theory and drew parallels with ECMAScript. Before reading of the current part, if it is necessary, I recommend reading the first part as in this article we will actively use the passed terminology. You can find the first part here: [ECMA-262-3 in detail. Chapter 7.1. OOP: The general theory](#).

ECMAScript OOP implementation

Having passed the way of highlights of the general theory, we at last have reached the ECMAScript itself. Now, when we know its OOP approach, let's make once again an accurate definition:

ECMAScript is an *object-oriented* programming language supporting *delegating inheritance based on prototypes*.

We begin the analysis from consideration of data types. And first it is necessary to notice that ECMAScript distinguishes entities on *primitive values* and *objects*. Therefore the phrase *"everything in JavaScript is an object"* sometimes arising in various articles, is not correct (is not full). Primitive values concern to a data of certain *types* which we should discuss in detail.

Data types

Though ECMAScript is a dynamic, weakly typed language with "duck" typing, and automatic type conversion, it nevertheless has certain data types. That is, at one moment, an object belongs to one concrete type.

Standard defines nine types, and only six are directly accessible in an ECMAScript program:

- Undefined
- Null
- Boolean
- String
- Number
- Object

Other three types are accessible only at implementation level (none of ECMAScript objects can have such type) and used by the specification for explaining behavior of some operations, for storing intermediate values and other. These are following types:

- Reference
- List
- Completion

Thus (in short overview), `Reference` type is used for an explanation of such operators as `delete`, `typeof`, `this` and other, and consists of a *base object* and a *property name*. `List` type describes behavior of the arguments list (in the `new` expression and function calls). `Completion` type in turn is used for an explanation of behavior `break`, `continue`, `return` and `throw` statements.

Primitive value types

Coming back to the six types used by ECMAScript programs, first five of them: `Undefined`, `Null`, `Boolean`, `String` and `Number` are types of *primitive values*.

Examples of primitive values:

```
var a = undefined;
var b = null;
var c = true;
var d = 'test';
var e = 10;
```

These values are represented in implementations directly on a low level. They are *not objects*, they do not have neither prototypes, nor constructors.

The `typeof` operator can be unintuitive if not properly understood. And one such example of that is with the value `null`. When `null` is supplied to the `typeof` operator, the result is `"object"` regardless of the fact that the type of `null` is specified as `Null`.

```
alert(typeof null); // "object"
```

And the reason is that the `typeof` operator returns the value taken from standard table which simply says: *"for `null` value string `"object"` should be returned"*.

Specification doesn't clarify this, however Brendan Eich (JavaScript inventor) noticed that `null` in contrast with `undefined`, is used in mostly where *objects appear*, i.e. is an essence closely related to objects (meaning the "empty" reference to an object, probably reserved a place for the future purposes). But, in some drafts, there was provided the document where this "phenomenon" was described as a usual bug. Also, this bug appeared in one of bug-trackers where Brendan Eich also participated; as a result it has been decided to leave `typeof null` as is, i.e. `"object"` though ECMA-262-3 standard defines type of `null` as `Null`.

Object type

In turn, the `Object` type (*do not confuse* with the `Object` *constructor*, we're talking now only about abstract types!) is the only type that represents ECMAScript objects.

Object is an unordered collection of key-value pairs.

The keys of objects are called *properties*. Properties are containers for primitive values and other objects. In case when properties contain functions as their values, they are called *methods*.

Example:

```
var x = { // object "x" with three properties: a, b, c
  a: 10, // primitive value
  b: {z: 100}, // object "b" with property z
  c: function () { // function (method)
    alert('method x.c');
  }
};

alert(x.a); // 10
alert(x.b); // [object Object]
```

```
alert(x.b.z); // 100
x.c(); // 'method x.c'
```

Dynamic nature

As we noted in chapter 7.1, objects in ES are fully *dynamic*. It means that we may add, modify or remove properties of objects at any time of program execution.

For example:

```
var foo = {x: 10};

// add new property
foo.y = 20;
console.log(foo); // {x: 10, y: 20}

// change property value to function
foo.x = function () {
  console.log('foo.x');
};

foo.x(); // 'foo.x'

// delete property
delete foo.x;
console.log(foo); // {y: 20}
```

Some properties cannot be modified — *read-only* properties or deleted — *non-configurable* properties. We'll consider these cases shortly in the section of property attributes.

Note, ES5 standardized *static* objects which cannot be extended with new properties and none of the properties can be modified or deleted. These are so-called *frozen* objects, which can be gotten by applying `Object.freeze(o)` method.

```
var foo = {x: 10};

// freeze the object
Object.freeze(foo);
console.log(Object.isFrozen(foo)); // true

// can't modify
foo.x = 100;

// can't extend
foo.y = 200;

// can't delete
delete foo.x;

console.log(foo); // {x: 10}
```

Also it's possible just to prevent extensions using `Object.preventExtensions(o)` method, and to control specific attributes with `Object.defineProperty(o)` method:

```
var foo = {x : 10};

Object.defineProperty(foo, "y", {
  value: 20,
  writable: false, // read-only
  configurable: false // non-configurable
});

// can't modify
foo.y = 200;

// can't delete
delete foo.y; // false

// prevent extensions
Object.preventExtensions(foo);
console.log(Object.isExtensible(foo)); // false

// can't add new properties
foo.z = 30;

console.log(foo); {x: 10, y: 20}
```

For details see [this chapter](<>).

Built-in, native and host objects

It is necessary to notice also that the specification distinguishes *native* objects, *built-in* objects and *host* objects.

Built-in and *native* objects are defined by the ECMAScript specification and the implementation, and a difference between them insignificant. *Native* objects are the all objects provided by ECMAScript implementation (some of them can be *built-in*, some can be created during the program execution, for example user-defined objects).

The *built-in* objects are a subtype of *native* objects which are *built into* the ECMAScript *prior to the beginning* of a program (for example, `parseInt`, `Math` etc.).

All *host objects* are objects provided by the host environment, typically a browser, and may include, for example, `window`, `alert`, etc.

Notice, that host objects may be implemented using ES itself and completely correspond to the specification's semantics. From this viewpoint, they can be named *(unofficially)* as *"native-host"* objects, though it's mostly a theoretical aspect. The specification however does not define any "native-host" concept.

Boolean, String and Number objects

Also for some primitives the specification defines special *wrapper objects*. These are following objects:

- Boolean-object
- String-object

- Number-object

Such objects are created with corresponding built in constructors and contain primitive value as one of internal properties. Object representation can be converted into primitive values and vice-versa.

Examples of the object values corresponding to primitive types:

```
var c = new Boolean(true);
var d = new String('test');
var e = new Number(10);

// converting to primitive
// conversion: ToPrimitive
// applying as a function, without "new" keyword
c = Boolean(c);
d = String(d);
e = Number(e);

// back to Object
// conversion: ToObject
c = Object(c);
d = Object(d);
e = Object(e);
```

Besides, there are also objects created by special built in constructors: `Function` (function objects constructor) `Array` (arrays constructor), `RegExp` (regular expressions constructor), `Math` (the mathematical module), `Date` (the constructor of dates), etc. Such objects are also values of type `Object` and their distinction from each other is managed by internal properties which we will discuss below.

Literal notations

For three object values: *object*, *array* and *regular expression* there are short notations which called accordingly an *object initialiser*, an *array initialiser* and a *regular expression literal*:

```
// equivalent to new Array(1, 2, 3);
// or array = new Array();
// array[0] = 1;
// array[1] = 2;
// array[2] = 3;
var array = [1, 2, 3];

// equivalent to
// var object = new Object();
// object.a = 1;
// object.b = 2;
// object.c = 3;
var object = {a: 1, b: 2, c: 3};

// equivalent to new RegExp("^\\d+$", "g")
var re = /^\d+$/g;
```

Notice, that in case of reassigning the name bindings — `Object`, `Array` or `RegExp` to some new objects, the semantics of subsequent using of the literal notations may vary in implementations. For example in the current Rhino implementation or in the old SpiderMonkey 1.7 appropriate literal notation will create an object corresponding to the *new* value of constructor name. In other implementations (including current Spider/TraceMonkey) semantics of the literal notations is not being changed even if constructor name is rebound to the new object:

```
var getClass = Object.prototype.toString;

Object = Number;

var foo = new Object;
alert([foo, getClass.call(foo)]); // 0, "[object Number]"

var bar = {};

// in Rhino, SpiderMonkey 1.7 - 0, "[object Number]"
// in other: still "[object Object]", "[object Object]"
alert([bar, getClass.call(bar)]);

// the same with Array name
Array = Number;

foo = new Array;
alert([foo, getClass.call(foo)]); // 0, "[object Number]"

bar = [];

// in Rhino, SpiderMonkey 1.7 - 0, "[object Number]"
// in other: still "", "[object Object]"
alert([bar, getClass.call(bar)]);

// but for RegExp, semantics of the literal
// isn't being changed in all tested implementations

RegExp = Number;

foo = new RegExp;
alert([foo, getClass.call(foo)]); // 0, "[object Number]"

bar = /(?!)/g;
alert([bar, getClass.call(bar)]); // /(?!)/g, "[object RegExp]"
```

Regular Expression Literal and RegExp Objects

Notice although, that in ES3 the two last cases with regular expressions being equivalent semantically, nevertheless differ. The *regexp literal* exists *only in one instance* and is created on parsing stage, while `RegExp` constructor creates always a *new object*. This can cause some issues with e.g. `lastIndex` property of regexp objects when regexp test is fail:

```
for (var k = 0; k < 4; k++) {
  var re = /ecma/g;
  alert(re.lastIndex); // 0, 4, 0, 4
  alert(re.test("ecmascript")); // true, false, true, false
```

```
}

// in contrast with

for (var k = 0; k < 4; k++) {
  var re = new RegExp("ecma", "g");
  alert(re.lastIndex); // 0, 0, 0, 0
  alert(re.test("ecmascript")); // true, true, true, true
}
```

Note, in ES5 this issue has been fixed and regexp literal also always creates a new object.

Associative arrays?

Often in various articles or discussions, JavaScript objects (and usually exactly those which created in declarative form — via the object initialiser — `{}`) are called *hash-tables* or simply — *hashes* (terms from Ruby or Perl), *associative arrays* (term from PHP), *dictionaries* (term from Python) etc.

Using of this terminology is a habit to concrete technology. Really, they are similar enough, and in respect of *"key-value"* pairs storage completely correspond to the theoretical *"associative array"* or *"hash tables"* data structures. Moreover, a hash table abstract data type may be and *usually is* used at implementation level.

However, although terminology is used to describe a conceptual way of thinking, it is not actually technically correct, regarding ECMAScript. As it has been noted, ECMAScript has only one object type and its "subtypes" in respect of a "key-value" pairs storage *do not differ from each other*. Therefore, there is no separated special term (hash or other) for that. Because any object regardless its internal properties can store these pairs:

```
var a = {x: 10};
a['y'] = 20;
a.z = 30;

var b = new Number(1);
b.x = 10;
b.y = 20;
b['z'] = 30;

var c = new Function('');
c.x = 10;
c.y = 20;
c['z'] = 30;

// etc. - with any object "subtype"
```

Moreover, objects in ECMAScript because of delegation can be nonempty, therefore the term "hash" also can be improper:

```
Object.prototype.x = 10;

var a = {}; // create "empty" "hash"

```

```
alert(a["x"]); // 10, but it's not empty
alert(a.toString); // function

a["y"] = 20; // add new pair to "hash"
alert(a["y"]); // 20

Object.prototype.y = 20; // and property into the prototype

delete a["y"]; // remove
alert(a["y"]); // but key and value are still here - 20
```

Notice, that ES5 standardized the ability to create objects *without prototypes* — that is, their prototype is set to `null`. It's achieved with using the `Object.create(null)` method. From this viewpoint such objects are simple hash-tables:

```
var aHashTable = Object.create(null);
console.log(aHashTable.toString); // undefined
```

Also, some properties can have specific getters/setters, so it can also confuse:

```
var a = new String("foo");
a['length'] = 10;
alert(a['length']); // 3
```

However, even if to consider that "hash" could have a "prototype" (as for example, in Ruby or Python — a class to which delegate hash-objects), in ECMAScript this terminology can also be improper because *there is no semantic differentiation* between *kinds of property accessors* (i.e. *dot* and *bracket* notations).

Also in ECMAScript concept of a "property" semantically is not separated into a "key", "array index", "method" or "property". Here all of them are *properties* which obey to the common law of reading/writing algorithm with examination of the prototype chain.

In the following example on Ruby we see this distinction in semantics and consequently there such terminology can differ:

```
a = {}
a.class # Hash

a.length # 0

# new "key-value" pair
a['length'] = 10;

# but semantics for the dot notation
# remains other and means access
# to the "property/method", but not to the "key"

a.length # 1

# and the bracket notation
# provides access to "keys" of a hash

a['length'] # 10
```

```

# we can augment dynamically Hash class
# with new properties/methods and they via
# delegation will be available for already created objects

class Hash
  def z
    100
  end
end

# a new "property" is available

a.z # 100

# but not a "key"

a['z'] # nil
```

ECMA-262-3 standard *does not define* concept of *"hash"* (and similar). However, if theoretical data structure is meant, it is possible to name objects so.

Type conversion

To convert an object into a primitive value the method `valueOf` can be used. As we noted, the call of the constructor (for certain types) as a function, i.e. *without* `new` operator performs conversion of object type to a primitive value. For this conversion exactly implicit call of the `valueOf` method is used:

```
var a = new Number(1);
var primitiveA = Number(a); // implicit "valueOf" call
var alsoPrimitiveA = a.valueOf(); // explicit

alert([
  typeof a, // "object"
  typeof primitiveA, // "number"
  typeof alsoPrimitiveA // "number"
]);
```

This method allows objects to participate in various operations, for example, in addition:

```
var a = new Number(1);
var b = new Number(2);

alert(a + b); // 3

// or even so

var c = {
  x: 10,
  y: 20,
  valueOf: function () {
    return this.x + this.y;
  }
```

```
};

var d = {
  x: 30,
  y: 40,
  // the same .valueOf
  // functionality as "c" object has,
  // borrow it:
  valueOf: c.valueOf
};

alert(c + d); // 100
```

The value of the `valueOf` method by default (if it is not overridden) can vary depending on object type. For some objects it returns the `this` value — for example, `Object.prototype.valueOf()`, for others — any calculated value, as e.g. `Date.prototype.valueOf()`, which returns the time of a date:

```
var a = {};
alert(a.valueOf() === a); // true, "valueOf" returned this value

var d = new Date();
alert(d.valueOf()); // time
alert(d.valueOf() === d.getTime()); // true
```

Also there is one more primitive representation of an object — a string representation. For this `toString` method is responsible, which in some operations is also applied automatically:

```
var a = {
  valueOf: function () {
    return 100;
  },
  toString: function () {
    return '__test';
  }
};

// in this operation
// toString method is
// called automatically
alert(a); // "__test"

// but here - the .valueOf() method
alert(a + 10); // 110

// but if there is no
// valueOf method, it
// will be replaced with the
//toString method
delete a.valueOf;
alert(a + 10); // "_test10"
```

The `toString` method defined on `Object.prototype` has special meaning. It returns the value of the internal `[[Class]]` property, which we'll discuss below.

Along with `ToPrimitive` conversion, there is also `ToObject` conversion which vice-versa converts the value to the *object type*.

One of explicit ways to call `ToObject` is to use built in `Object` constructor as a function (though for some types using of `Object` with the `new` operator is also possible):

```
var n = Object(1); // [object Number]
var s = Object('test'); // [object String]

// also for some types it is
// possible to call Object with new operator
var b = new Object(true); // [object Boolean]

// but applied without arguments,
// new Object creates a simple object
var o = new Object(); // [object Object]

// in case if argument for Object function
// is already object value,
// it simply returns
var a = [];
alert(a === new Object(a)); // true
alert(a === Object(a)); // true
```

Regarding calls of the built in constructors with the `new` and without `new` operator, there is no the general rule and it depends on the constructor. For example `Array` or `Function` constructors produce *the same* results when are called as a constructor (with `new`) and as a simple function (without `new`):

```
var a = Array(1, 2, 3); // [object Array]
var b = new Array(1, 2, 3); // [object Array]
var c = [1, 2, 3]; // [object Array]

var d = Function(''); // [object Function]
var e = new Function(''); // [object Function]
```

There are also explicit and implicit type casting when some operators are applied:

```
var a = 1;
var b = 2;

// implicit
var c = a + b; // 3, number
var d = a + b + '5' // "35", string

// explicit
var e = '10'; // "10", string
var f = +e; // 10, number
var g = parseInt(e, 10); // 10, number

// etc.
```

Property attributes

All properties can have a number of attributes:

- `{ReadOnly}` — attempt to write value to the property is ignored; however, ReadOnly-properties can be changed by host-environment actions, therefore ReadOnly — does not mean "constant value";
- `{DontEnum}` — the property is not enumerable by a `for..in` loop;
- `{DontDelete}` — action of the `delete` operator applied to the property is ignored;
- `{Internal}` — the property is internal, it has no name and is used only on implementation level; such properties are not accessible to the ECMAScript program.

Note, in ES5 `{ReadOnly}`, `{DontEnum}` and `{DontDelete}` are renamed accordingly into the `[[Writable]]`, `[[Enumerable]]` and `[[Configurable]]` and can be manually managed via the `Object.defineProperty` and similar methods.

```
var foo = {};

Object.defineProperty(foo, "x", {
  value: 10,
  writable: true, // aka {ReadOnly} = false
  enumerable: false, // aka {DontEnum} = true
  configurable: true // {DontDelete} = false
});

console.log(foo.x); // 10

// attributes set is called a descriptor
var desc = Object.getOwnPropertyDescriptor(foo, "x");

console.log(desc.enumerable); // false
console.log(desc.writable); // true
// etc.
```

Internal properties and methods

Objects also can have a number of internal properties which are a part of implementation and inaccessible for ECMAScript programs directly (however as we will see below, some implementations allow the access to some such properties). These properties by the convention are enclosed with double square brackets — `[[ ]]`.

We will touch some of them (obligatory for all objects); description of other properties can be found in the specification.

Each object should implement the following internal properties and methods:

- `[[Prototype]]` — the prototype of this object (it will be considered below in detail);
- `[[Class]]` — a string representation of object's *kind* (for example, `Object`, `Array`, `Function`, etc.); it is used to distinguish the objects;
- `[[Get]]` — a method of getting the property's value;
- `[[Put]]` — a method of setting the property's value;
- `[[CanPut]]` — checks whether writing to the property is possible;
- `[[HasProperty]]` — checks whether the object has already this property;
- `[[Delete]]` — removes the property from the object;
- `[[DefaultValue]]` — returns a primitive value corresponding with the object (for getting this value

the `valueOf` method is called; for some objects, `TypeError` exception can be thrown).

To get the `[[Class]]` property from ECMAScript programs is possible indirectly via the `Object.prototype.toString()` method. This method should return the following string: `"[object " + [[Class]] + "]"`. For example:

```
var getClass = Object.prototype.toString;

getClass.call({}); // [object Object]
getClass.call([]); // [object Array]
getClass.call(new Number(1)); // [object Number]
// etc.
```

This feature is often used to check the kind of an object, however, it is necessary to note that by the specification internal `[[Class]]` property of *hosts-objects* can be *any*, including values of the `[[Class]]` property of the built in objects, that in theory does not make such checks 100% proved. For example, `[[Class]]` property of the `document.childNodes.item(...)` method in older IE returns `"String"` (in other implementations, `"Function"` is returned):

```
// in older IE - "String", in other - "Function"
alert(getClass.call(document.childNodes.item));
```

Constructor

So, as we mentioned above, objects in ECMAScript are created via, so-called, *constructors*.

Constructor is a function that *creates* and *initializes* the newly created object.

For *creation (memory allocation)* the `[[Construct]]` internal method of a constructor function is responsible. The behavior of this internal method is specified and all constructor functions uses this method to allocate memory for new object.

And *initialization* is managed by calling the function in context of newly created object. For this already internal `[[Call]]` method of the constructor function is responsible.

Note, that from user-code only the initialization phase is accessible. Though, even from initialization we can return *different object* ignoring `this` object which was created at the first stage:

```
function A() {
  // update newly created object
  this.x = 10;
  // but return different object
  return [1, 2, 3];
}

var a = new A();
console.log(a.x, a); undefined, [1, 2, 3]
```

Referencing to algorithm of creation of Function objects discussed in the Chapter 5. Functions, we see that function is a native object which among other properties has this internal `[[Construct]]` and `[[Call]]` properties and also explicit `prototype` property — the reference to a prototype of the future objects (notice, `NativeObject` here and below is my pseudo-code naming convention for "native object" concept from ECMA-262-3, but not the built-in constructor).

```
F = new NativeObject();

F.[[Class]] = "Function"

.... // other properties

F.[[Call]] = <reference to function> // function itself

F.[[Construct]] = internalConstructor // general internal construc

.... // other properties

// prototype of objects created by the F constructor
__objectPrototype = {};
__objectPrototype.constructor = F // {DontEnum}
F.prototype = __objectPrototype
```

Thus `[[Call]]` besides the `[[Class]]` property (which equals to `"Function"`) is the main in respect of objects distinguishing. Therefore the objects having internal `[[Call]]` property are called as *functions*. The `typeof` operator for such objects returns `"function"` value. However, it mostly relates to *native objects*, in case of *host callable objects*, the `typeof` operator (no less than `[[Class]]` property) of some implementations can return other value: for example, `window.alert(...)` in IE:

```
// in IE - "Object", "object", in other - "Function", "function"
alert(Object.prototype.toString.call(window.alert));
alert(typeof window.alert); // "Object"
```

The internal `[[Construct]]` method is activated by the `new` operator applied to the constructor function. As we said this method is responsible for memory allocation and creation of the object. If there are no arguments, call parenthesis of constructor function can be omitted:

```
function A(x) { // constructor A
  this.x = x || 10;
}

// without arguments, call
// brackets can be omitted
var a = new A; // or new A();
alert(a.x); // 10

// explicit passing of
// x argument value
var b = new A(20);
alert(b.x); // 20
```

And as also we know, [this value](#) inside the constructor (at initialization phase) is set to the *newly created object*.

Let's consider the algorithm of objects creation.

Algorithm of objects creation

The behavior of the internal `[[Construct]]` method can be described as follows:

```
F.[[Construct]](initialParameters):

O = new NativeObject();

// property [[Class]] is set to "Object", i.e. simple object
O.[[Class]] = "Object"

// get the object on which
// at the moment references F.prototype
var __objectPrototype = F.prototype;

// if __objectPrototype is an object, then:
O.[[Prototype]] = __objectPrototype
// else:
O.[[Prototype]] = Object.prototype;
// where O.[[Prototype]] is the prototype of the object

// initialization of the newly created object
// applying the F.[[Call]]; pass:
// as this value – newly created object - O,
// arguments are the same as initialParameters for F
R = F.[[Call]](initialParameters); this === O;
// where R is the returned value of the [[Call]]
// in JS view it looks like:
// R = F.apply(O, initialParameters);

// if R is an object
return R
// else
return O
```

Note two major features:

First, the *prototype* of the created object is taken from the `prototype` property of a function on the *current* moment (it means that the prototype of two created objects from one constructor can vary since the `prototype` property of a function can also vary).

Secondly, as we have mentioned above, if at object initialization the `[[Call]]` has returned an *object*, exactly it is used as the *result* of the whole `new` expression:

```
function A() {}
A.prototype.x = 10;

var a = new A();
alert(a.x); // 10 – by delegation, from the prototype

// set .prototype property of the
```

```
// function to new object; why explicitly
// to define the .constructor property,
// will be described below
A.prototype = {
  constructor: A,
  y: 100
};

var b = new A();
// object "b" has new prototype
alert(b.x); // undefined
alert(b.y); // 100 - by delegation, from the prototype

// however, prototype of the "a" object
// is still old (why - we will see below)
alert(a.x); // 10 - by delegation, from the prototype

function B() {
  this.x = 10;
  return new Array();
}

// if "B" constructor had not return
// (or was return this), then this-object
// would be used, but in this case - an array
var b = new B();
alert(b.x); // undefined
alert(Object.prototype.toString.call(b)); // [object Array]
```

Let's consider a prototype deeply in detail.

Prototype

Every object has a prototype (exceptions can be with some system objects). Communication with a prototype is organized via the *internal*, implicit and inaccessible directly `[[Prototype]]` property. A prototype can be either an *object*, or the `null` value.

Property `constructor`

In the above example there are two important points. The first relates to `constructor` property of the function's `prototype` property.

As we can see in algorithm of function objects creation, `constructor` property is set to function's `prototype` property at function *creation*. The value of this property is the *circular reference* to the function *itself*:

```
function A() {}
var a = new A();
alert(a.constructor); // function A() {}, by delegation
alert(a.constructor === A); // true
```

Often in this case there is a misunderstanding — `constructor` property is *incorrectly* treated as own property of the created object. However as we have seen, this property belongs to a *prototype* and is accessible to object via *inheritance*.

Via the *inherited* `constructor` property instances can *indirectly* get the reference to the prototype object:

```
function A() {}
A.prototype.x = new Number(10);

var a = new A();
alert(a.constructor.prototype); // [object Object]

alert(a.x); // 10, via delegation
// the same as a.[[Prototype]].x
alert(a.constructor.prototype.x); // 10

alert(a.constructor.prototype.x === a.x); // true
```

Notice though, that both `constructor` and `prototype` properties of the function can be *redefined* after the object is created. In this case the object looses the reference via the mechanism above.

If we *add* new or *modifiy* existing property in the original prototype via the function's `prototype` property, instances will see the newly added properties.

However, if we *change* function's `prototype` property *completely* (via *assigning* a new object), the reference to the *original* constructor (as well as to the original prototype) *is lost*. This is because we create the new object which *does not have* `constructor` property:

```
function A() {}
A.prototype = {
  x: 10
};

var a = new A();
alert(a.x); // 10
alert(a.constructor === A); // false!
```

Therefore, this reference should be restored manually:

```
function A() {}
A.prototype = {
  constructor: A,
  x: 10
};

var a = new A();
alert(a.x); // 10
alert(a.constructor === A); // true
```

Notice though that the *restored manually* `constructor` property, in contrast with the *lost original*, *has no* attribute `{DontEnum}` and, as consequence, is enumerable in the `for..in` loop over the `A.prototype`.

In ES5 was introduced the ability of controlling enumerable state of properties via the `[[Enumerable]]` attribute.

```
var foo = {x: 10};

Object.defineProperty(foo, "y", {
  value: 20,
  enumerable: false // aka {DontEnum} = true
});

console.log(foo.x, foo.y); // 10, 20

for (var k in foo) {
  console.log(k); // only "x"
}

var xDesc = Object.getOwnPropertyDescriptor(foo, "x");
var yDesc = Object.getOwnPropertyDescriptor(foo, "y");

console.log(
  xDesc.enumerable, // true
  yDesc.enumerable  // false
);
```

Explicit `prototype` and implicit `[[Prototype]]` properties

Often prototype of an object is incorrectly confused with explicit reference to the prototype via the function's `prototype` property. Yes, really, it references to the *same object*, as object's `[[Prototype]]` property:

```
a.[[Prototype]] ----> Prototype <---- A.prototype
```

Moreover, `[[Prototype]]` of an instance gets its value from exactly the `prototype` property of the constructor — at object's creation.

However, replacing `prototype` property of the constructor *does not affect* the prototype of *already created objects*. It's *only* the `prototype` property of the constructor that is changed! It means that *new objects* will have a *new prototype*. But *already created* objects (before the `prototype` property was changed), have reference to the *old prototype* and this reference *cannot be changed already*:

```
// was before changing of A.prototype
a.[[Prototype]] ----> Prototype <---- A.prototype

// became after
A.prototype ----> New prototype // new objects will have this protc
a.[[Prototype]] ----> Prototype // reference to old prototype
```

Example:

```
function A() {}
A.prototype.x = 10;

```

```
var a = new A();
alert(a.x); // 10

A.prototype = {
  constructor: A,
  x: 20
  y: 30
};

// object "a" delegates to
// the old prototype via
// implicit [[Prototype]] reference
alert(a.x); // 10
alert(a.y) // undefined

var b = new A();

// but new objects at creation
// get reference to new prototype
alert(b.x); // 20
alert(b.y) // 30
```

Therefore, sometimes arising statements in articles on JavaScript claiming that *"dynamic changing of the prototype will affect all objects and they will have that new prototype"* is *incorrect*. New prototype will have *only new* objects which will be created after this changing.

The main rule here is: the object's prototype is set at the moment of object's *creation* and after that *cannot be changed* to new object. Using the explicit `prototype` reference from the constructor if it still refers to the *same* object, it is possible *only* to *add new* or *modify* existing properties of the object's prototype.

Non-standard `__proto__` property

However, some implementations, for example, SpiderMonkey, provide *explicit* reference to object's prototype via the non-standard `__proto__` property:

```
function A() {}
A.prototype.x = 10;

var a = new A();
alert(a.x); // 10

var __newPrototype = {
  constructor: A,
  x: 20,
  y: 30
};

// reference to new object
A.prototype = __newPrototype;

var b = new A();
```

```
alert(b.x); // 20
alert(b.y); // 30

// "a" object still delegates
// to the old prototype
alert(a.x); // 10
alert(a.y); // undefined

// change prototype explicitly
a.__proto__ = __newPrototype;

// now "a" object references
// to new object also
alert(a.x); // 20
alert(a.y); // 30
```

Note, ES5 introduced `Object.getPrototypeOf(O)` method, which directly returns the `[[Prototype]]` property of an object — the original prototype of the instance. However, in contrast with `__proto__`, being only a *getter*, it does not allow to set the prototype.

```
var foo = {};
Object.getPrototypeOf(foo) == Object.prototype; // true
```

Object is independent from its constructor

Since the prototype of an instance is independent from the constructor and the `prototype` property of the constructor, the constructor after its main purpose — creation of the object — can be *removed*. The prototype object will continue to exist, being referenced via the `[[Prototype]]` property:

```
function A() {}
A.prototype.x = 10;

var a = new A();
alert(a.x); // 10

// set "A" to null - explicit
// reference on constructor
A = null;

// but, still possible to create
// objects via indirect reference
// from other object if
// .constructor property has not been changed
var b = new a.constructor();
alert(b.x); // 10

// remove both implicit references
delete a.constructor.prototype.constructor;
delete b.constructor.prototype.constructor;

// it is not possible to create objects
// of "A" constructor anymore, but still
```

```
// there are two such objects which
// still have reference to their prototype
alert(a.x); // 10
alert(b.x); // 10
```

Feature of `instanceof` operator

With the explicit reference to a prototype — via the `prototype` property of the constructor, the work of the `instanceof` operator is related.

This operator works *exactly* with the *prototype chain* of an object but not with the constructor itself. Take this into account, since there is often misunderstanding at this place. That is, when there is a check:

```
if (foo instanceof Foo) {
  ...
}
```

it *does not mean* the check whether the object `foo` is *created* by the `Foo` constructor!

All the `instanceof` operator does is only takes the value of the `Foo.prototype` property and checks its *presence* in the *prototype chain* of `foo`, starting from the `foo.[[Prototype]]`. The `instanceof` operator is activated by the internal [[HasInstance]] method of the constructor.

Let’s see it on the example:

```
function A() {}
A.prototype.x = 10;

var a = new A();
alert(a.x); // 10

alert(a instanceof A); // true

// if set A.prototype
// to null...
A.prototype = null;

// ...then "a" object still
// has access to its
// prototype - via a.[[Prototype]]
alert(a.x); // 10

// however, instanceof operator
// can't work anymore, because
// starts its examination from the
//prototype property of the constructor
alert(a instanceof A); // error, A.prototype is not an object
```

On the other hand, it is possible to create object by one constructor, but `instanceof` will return `true` on check with *another* constructor. All that is necessary is to set object’s `[[Prototype]]` property and `prototype` property of the constructor to the same object:

```
function B() {}
var b = new B();

alert(b instanceof B); // true

function C() {}

var __proto = {
  constructor: C
};

C.prototype = __proto;
b.__proto__ = __proto;

alert(b instanceof C); // true
alert(b instanceof B); // false
```

Prototype as a storage for methods and shared properties

The most useful application of the prototype in ECMAScript is the storage of *methods*, *default state* and *shared properties* of objects.

Indeed, objects can have their own *states*, but methods are usually the same. Therefore, methods, for optimization of a memory usage, are usually defined in the prototype. It means that all instances created by this constructor, always *share* the *same* method.

```
function A(x) {
  this.x = x || 100;
}

A.prototype = (function () {

  // initializing context,
  // use additional object

  var _someSharedVar = 500;

  function _someHelper() {
    alert('internal helper: ' + _someSharedVar);
  }

  function method1() {
    alert('method1: ' + this.x);
  }

  function method2() {
    alert('method2: ' + this.x);
    _someHelper();
  }

  // the prototype itself
  return {
    constructor: A,
```

```
    method1: method1,
    method2: method2
  };

})();

var a = new A(10);
var b = new A(20);

a.method1(); // method1: 10
a.method2(); // method2: 10, internal helper: 500

b.method1(); // method1: 20
b.method2(); // method2: 20, internal helper: 500

// both objects are use
// the same methods from
// the same prototype
alert(a.method1 === b.method1); // true
alert(a.method2 === b.method2); // true
```

Reading and writing properties

As we mentioned, reading and writing of properties are managed by the internal methods `[[Get]]` and `[[Put]]`. The methods are activated by *property accessors* — dot notation or brackets notation:

```
// write
foo.bar = 10; // [[Put]] is called

console.log(foo.bar); // 10, [[Get]] is called
console.log(foo['bar']); // the same
```

Let’s show the work of these methods as a pseudo-code.

`[[Get]]` method

The `[[Get]]` method considers the properties from the *prototype chain* of object as well. Therefore properties of a prototype are accessible to object as own.

```
O.[[Get]](P):

// if there is own
// property, return it
if (O.hasOwnProperty(P)) {
  return O.P;
}

// else, analyzing prototype
var __proto = O.[[Prototype]];

// if there is no prototype (it is,
// possible e.g. in the last link of the
```

```
// chain - Object.prototype.[[Prototype]],
// which is equal to null),
// then return undefined;
if (__proto === null) {
  return undefined;
}

// else, call [[Get]] method recursively -
// now for prototype; i.e. go through prototype
// chain: try to find property in the
// prototype, after that - in a prototype of
// the prototype and so on, until
// [[Prorotype]] will be equal to null
return __proto.[[Get]](P)
```

Note, since the `[[Get]]` method in one of cases can return `undefined`, checks on variable presence, like the following are possible:

```
if (window.someObject) {
  ...
}
```

Here, property `someObject` is not found in `window`, then in its prototype, in the prototype of the prototype etc., and in this case, by the algorithm, `undefined` value is returned.

Notice, that for *exactly presence* the `in` operator is responsible. It also considers the prototype chain:

```
if ('someObject' in window) {
  ...
}
```

It helps to avoid cases when, for example, `someObject` can be equal to `false` and the first check does not pass even if `someObject` exists.

[`[[Put]]` method]()

The `[[Put]]` method in contrast creates or updates an *own* property of the object and *shadows* the property with the same name from the prototype.

```
O.[[Put]](P, V):

// if we can't write to
// this property then exit
if (!O.[[CanPut]](P)) {
  return;
}

// if object doesn't have such own,
// property, then create it; all attributes
// are empty (set to false)
if (!O.hasOwnProperty(P)) {
  createNewProperty(O, P, attributes: {
```

```
    ReadOnly: false,
    DontEnum: false,
    DontDelete: false,
    Internal: false
  });
}

// set the value;
// if property existed, its
// attributes are not changed
O.P = V

return;
```

For example:

```
Object.prototype.x = 100;

var foo = {};
console.log(foo.x); // 100, inherited

foo.x = 10; // [[Put]]
console.log(foo.x); // 10, own

delete foo.x;
console.log(foo.x); // again 100, inherited
```

Notice, it's *not* possible to *shadow inherited read-only* property. Result of an assignment is just ignored. This is controlled by the `[[CanPut]]` internal method; see [8.6.2.3](#) of ES3.

```
// For example, property "length" of
// string objects is read-only; let's make a
// string as a prototype of our object and try
// to shadow the "length" property

function SuperString() {
  /* nothing */
}

SuperString.prototype = new String("abc");

var foo = new SuperString();

console.log(foo.length); // 3, the length of "abc"

// try to shadow
foo.length = 5;
console.log(foo.length); // still 3
```

In [strict mode](#) of ES5 an attempt to shadow a non-writable property [results](#) a `TypeError`.

Property accessors

That's said, internal methods `[[Get]]` and `[[Put]]` are activated by *property accessors* which in ECMAScript are available via the *dot notation*, or via the *bracket notation*. The dot notation is used when the property name is a valid identifier name and in advance known, bracket notation allows forming names of properties dynamically.

```
var a = {testProperty: 10};

alert(a.testProperty); // 10, dot notation
alert(a['testProperty']); // 10, bracket notation

var propertyName = 'Property';
alert(a['test' + propertyName]); // 10, bracket notation with dynan
```

There is one important feature — property accessor always calls `ToObject` conversion for the object standing on left hand side from the property accessor. And because of this implicit conversion it is possible *roughly speaking* to say that *"everything in JavaScript is an object"* (however as we already know — of course not everything since there are also primitive things).

If we use property accessor with a *primitive value*, we just create *intermediate wrapper object* with corresponding value. After the work is finished, this wrapper is *removed*.

Example:

```
var a = 10; // primitive value

// but, it has access to methods,
// just like it would be an object
alert(a.toString()); // "10"

// moreover, we can even
// (try) to create a new
// property in the "a" primitive calling [[Put]]
a.test = 100; // seems, it even works

// but, [[Get]] doesn't return
// value for this property, it returns
// by algorithm - undefined
alert(a.test); // undefined
```

So, why in this example "primitive" value `a` has access to the `toString` method, but has no to the newly created `test` property?

The answer is simple:

First, as we said, after the property accessor is applied, it is already *not a primitive*, but the *intermediate object*. In this case *new Number(a)* is used, which via delegation finds the `toString` method in the prototype chain:

```
// Algorithm of evaluating a.toString():

1. wrapper = new Number(a);
2. wrapper.toString(); // "10"
```

```
3. delete wrapper;
```

Next, `[[Put]]` method also creates *its own wrapper object* when evaluating the `test` property:

```
// Algorithm of evaluating a.test = 100:

1. wrapper = new Number(a);
2. wrapper.test = 100;
3. delete wrapper;
```

We see that in step 3 the wrapper is *removed* and its *newly created* `test` property *is of course also* — with removing the object itself.

Then again `[[Get]]` is called where the property accessor creates *again new wrapper* which of course *does not known* anything about any `test` property:

```
// Algorithm of evaluating a.test:

1. wrapper = new Number(a);
2. wrapper.test; // undefined
```

That is the reference to properties/methods from a *primitive* value makes sense only for *reading* the properties. Also if any of primitive values often uses the access to properties, for economy of time resources, there is a sense directly to replace it with an object representation. And on the contrary — if values participate only in some small calculations which are not demanding the access to properties then more efficiently primitive values can be used.

Inheritance

As we know, ECMAScript uses *delegating inheritance based on prototypes*.

Chaining, prototypes generate already mentioned *prototype chain*.

Actually, all work for implementing delegation and the analysis of a prototype chain is reduced to the work of the mentioned above `[[Get]]` method.

If you completely understand the simple algorithm of the `[[Get]]` method, the question on inheritance in JavaScript will disappear by itself and the answer to it will become clear.

Often on forums when the talk comes about inheritance in JavaScript, I show as an example only one line of ECMAScript code which very exactly and accurate describes object structure of the language and shows delegation based inheritance. Indeed, we can do not create any constructors or objects but the whole language *is already full of inheritance*. The line of code is very simple:

```
alert(1..toString()); // "1"
```

Now, when we know the algorithm of the `[[Get]]` method and property accessors, we can see what happens here:

1. First, from a primitive value `1` the *wrapper object* as `new Number(1)` is created;

2. Then the *inherited* method `toString` is called from this *wrapper*.

Why the inherited? Because objects in ECMAScript can have *own* properties, and the created wrapper object, in this case, *has no* own `toString` method. Therefore, it *inherits* it from a prototype, i.e. `Number.prototype`.

Notice the subtle case of the syntax. Two dots in the example above *is not an error*. The first dot is used for *fractional part of a number*, and the second one is already a *property accessor*:

```
1.toString(); // SyntaxError!

(1).toString(); // OK

1 .toString(); // OK (space after 1)

1..toString(); // OK

1['toString'](); // OK
```

Prototype chain

Let's show how to create these prototype chains for the *user-defined* objects. It is quite simple:

```
function A() {
  alert('A.[[Call]] activated');
  this.x = 10;
}
A.prototype.y = 20;

var a = new A();
alert([a.x, a.y]); // 10 (own), 20 (inherited)

function B() {}

// the easiest variant of prototypes
// chaining is setting child
// prototype to new object created,
// by the parent constructor
B.prototype = new A();

// fix .constructor property, else it would be A
B.prototype.constructor = B;

var b = new B();
alert([b.x, b.y]); // 10, 20, both are inherited

// [[Get]] b.x:
// b.x (no) -->
// b.[[Prototype]].x (yes) - 10

// [[Get]] b.y
// b.y (no) -->
// b.[[Prototype]].y (no) -->
// b.[[Prototype]].[[Prototype]].y (yes) - 20

```

```
// where b.[[Prototype]] === B.prototype,
// and b.[[Prototype]].[[Prototype]] === A.prototype
```

This approach has two features.

First, `B.prototype` will contain `x` property. At first glance, seems that it is not correct, since `x` property is defined in `A` as *own* and is expected to be *own* as well in objects of the `B` constructor.

In a case of prototypal inheritance though it is normal situation, since the descendant object, if has no such own property delegates to a prototype. The idea behind this is that probably, objects created by the `B` constructor *do not* need `x` property. In contrast, in the class based model, all properties are *copied* to the class-descendant.

However, if nevertheless it is needed (emulating class-based approach) that `x` property be *own* for the objects created by `B` constructor, there are some techniques for this, one of which we will show below.

Secondly, that is already not a feature but the *disadvantage* — the code of the constructor is also executed when the descendant prototype is created. We can see that the message `"A.[[Call]] activated"` is shown *twice* — when the object created by the `A` constructor is used for `B.prototype` and also at creation of object `a` object itself!

A more critical example is a thrown exception in the parent constructor: perhaps, for the *real* objects created by this constructor such checks are needed, but obviously, the same case is completely unacceptable with using these parent objects as prototypes:

```
function A(param) {
  if (!param) {
    throw 'Param required';
  }
  this.param = param;
}
A.prototype.x = 10;

var a = new A(20);
alert([a.x, a.param]); // 10, 20

function B() {}
B.prototype = new A(); // Error
```

Besides, heavy calculations in the parent constructor can also be considered as disadvantage of this approach.

To solve these "features" and issues, today programmers use standard pattern for chaining the prototypes, which we show below. The main goal of this trick consists in creation of the *intermediate wrapper constructor* which chains the needed prototypes.

```
function A() {
  alert('A.[[Call]] activated');
  this.x = 10;
}
A.prototype.y = 20;

```

```
var a = new A();
alert([a.x, a.y]); // 10 (own), 20 (inherited)

function B() {
  // or simply A.apply(this, arguments)
  B.superproto.constructor.apply(this, arguments);
}

// inheritance: chaining prototypes
// via creating empty intermediate constructor
var F = function () {};
F.prototype = A.prototype; // reference
B.prototype = new F();
B.superproto = A.prototype; // explicit reference to ancestor prot

// fix .constructor property, else it would be A
B.prototype.constructor = B;

var b = new B();
alert([b.x, b.y]); // 10 (own), 20 (inherited)
```

Notice how we create own property `x` on `b` instance: we call parent constructor via the `B.superproto.constructor` reference in context of newly created object.

We have fixed also the issue with non-needed call of the parent constructor for creating the descendant prototype. Now the message `"A.[[Call]] activated"` is shown when is needed.

And for not to repeat every time the same actions of prototypes chaining (creation of the intermediate constructor, setting this `superproto` sugar, restoring the original `constructor` etc.), this template can be encapsulated in the convenient util function, which purpose is to chain prototypes regardless the concrete names of constructors:

```
function inherit(child, parent) {
  var F = function () {};
  F.prototype = parent.prototype
  child.prototype = new F();
  child.prototype.constructor = child;
  child.superproto = parent.prototype;
  return child;
}
```

Accordingly, inheritance:

```
function A() {}
A.prototype.x = 10;

function B() {}
inherit(B, A); // chaining prototypes

var b = new B();
alert(b.x); // 10, found in the A.prototype
```

There are many variations of such wrappers (in respect of syntax); however, all of them are reduced to the actions described above.

For example, we can optimize the previous wrapper if we will put intermediate constructor outside (thus, only one function will be created), thereby, reusing it:

```
var inherit = (function(){
  function F() {}
  return function (child, parent) {
    F.prototype = parent.prototype;
    child.prototype = new F;
    child.prototype.constructor = child;
    child.superproto = parent.prototype;
    return child;
  };
})();
```

Since the real prototype of an object is the `[[Prototype]]` property, it means that the `F.prototype` can be easily changed and reused, because `child.prototype`, being created via `new F`, will get its `[[Prototype]]` from the the *current* value of `child.prototype`:

```
function A() {}
A.prototype.x = 10;

function B() {}
inherit(B, A);

B.prototype.y = 20;

B.prototype.foo = function () {
  alert("B#foo");
};

var b = new B();
alert(b.x); // 10, is found in A.prototype

function C() {}
inherit(C, B);

// and using our "superproto" sugar
// we can call parent method with the same name

C.prototype.foo = function () {
  C.superproto.foo.call(this);
  alert("C#foo");
};

var c = new C();
alert([c.x, c.y]); // 10, 20

c.foo(); // B#foo, C#foo
```

Note, that ES5 has standardized this util function for better prototypes chaining. It is the

`Object.create` method.

Simplified version in ES3 can nearly be implemented in the following way:

```
Object.create ||
Object.create = function (parent, properties) {
  function F() {}
  F.prototype = parent;
  var child = new F;
  for (var k in properties) {
    child[k] = properties[k].value;
  }
  return child;
}
```

Usage:

```
var foo = {x: 10};
var bar = Object.create(foo, {y: {value: 20}});
console.log(bar.x, bar.y); // 10, 20
```

For details see [this chapter](#).

Also, all existing variations of imitations of *"classical inheritance in JS"* are based on this principle. Now we see, that in fact it is even not an "imitation of class based inheritance", but simply a *convenient code reuse for chaining prototypes*.

Notice: in ES6 the concept of a "class" is standardized, and is implemented as exactly a syntactic sugar over the constructor functions as described above. From this viewpoint prototype chains become as an implementation detail of the class-based inheritance:

```
// ES6
class Foo {
  constructor(name) {
    this._name = name;
  }

  getName() {
    return this._name;
  }
}

class Bar extends Foo {
  getName() {
    return super.getName() + ' Doe';
  }
}

var bar = new Bar('John');
console.log(bar.getName()); // John Doe
```

Conclusion

This article has turned out big enough and detailed. I hope that its material is useful and has dispelled some doubts regarding ECMAScript. If you have any questions or additions, they as always can be discussed in comments.

Additional literature

- 4.2 — Language Overview;
- 4.3 — Definitions;
- 7.8.5 — Regular Expression Literals;
- 8 — Types;
- 9 — Type Conversion;
- 11.1.4 — Array Initialiser;
- 11.1.5 — Object Initialiser;
- 11.2.2 — The new Operator;
- 13.2.1 — [[Call]];
- 13.2.2 — [[Construct]];
- 15 — Native ECMAScript Objects.

Translated by: Dmitry Soshnikov with additions by Garrett Smith.
Published on: 2010-03-04

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

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

50 Comments:

#permalink

1.  Nicolas
   20. March 2010 at 12:18

   Great article!

BTW, I think I've found a typo that I described here: http://gist.github.com/338559

Can't wait for the Functions article to be translated!

BR,

Nico.

2.  [Dmitry A. Soshnikov](#) [#permalink](#)
   20. March 2010 at 13:26

   @**Nicolas**

   Thanks, yeah, it was a typo of course; fixed.

   Yes, I'm planning translations of all chapters and now have already started translation of the "Chapter 4. Scope chain". After that "Chapter 5. Functions" will follow.

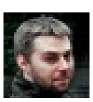
   Dmitry.

3. newbee [#permalink](#)
   28. March 2010 at 01:47

   Hi Dmitry,

   Maybe you can use "Google Translate" as a help when translating these great articles:

   http://translate.google.com/translate?hl=en&sl=ru&tl=en&u=http%3A%2F%2Fdmitrysoshnikov.com%2Fecmascript%2Fru-chapter-5-functions%2F

4.  [Dmitry A. Soshnikov](#) [#permalink](#)
   28. March 2010 at 12:54

   @**newbee**

   Hi, yeah it could be taken as a basis, but unfortunately all automatic translators don't know special terminology and cannot produce correct sentences regarding exactly specific technology. So even after basic translation it is required to work on every sentence.

   Dmitry.

5. John Merge [#permalink](#)

7. April 2010 at 21:44

Great article, as usual.
I can not explain how much I appreciate your work! You should write a book – you already have a lot of wonderful stuff. It will definitely become best-seller.

Thanks, and keep in this way!

John

#permalink

6.  Dmitry A. Soshnikov
7. April 2010 at 22:37

@**John Merge**

Thanks, John. And I am glad to see that quantity of interested in deep JS programmers are getting more and more.

> You should write a book – you already have a lot of wonderful stuff. It will definitely become best-seller.

Yes, I have such plans. Now negotiations are continuing with several publishers. The audience is limited to already experienced in JS (and in programming in a whole) programmers, so it is very important to choose a publisher correctly.

I think this book will be useful for every professional ECMAScript programmer.

Dmitry.

#permalink

7. John Merge
7. April 2010 at 22:39

Dmitry,

Your "inherit" function differs from those, used in some JavaScript libraries, for example YUI. Here is it:
In your code:

```
child. super = parent;
```

In YUI they do the following:

```
child. super = parent.prototype;
```

Take a look here (search for "extend"): http://github.com/yui/yui2/blob/master/src/yahoo/js/Lang.js

This is YUI2, but it is the same in YUI3: http://github.com/yui/yui3/blob/master/src/oop/js/oop.js

What is your explanation?

Thanks,
John

8.  [Dmitry A. Soshnikov](#) [#permalink](#)
8. April 2010 at 12:10

@**John Merge**

> Your “inherit” function differs from those, used in some JavaScript libraries

Yes, maybe; as I mentioned in the article, there are a lot of such implementations of code reuse in respect of chaining prototypes.

So, the reference to parent “class” can be easily set not to parent constructor itself, but to the prototype of an object (i.e. via explicit reference — *parent.prototype*). And it’s even better, since there is no need to repeat every time intermediate “.prototype.” property on accessing parent methods/properties. And at the same time we still have access to constructor function itself via “_super.constructor”.

I know that this approach is used in many implementations and frameworks, but in this article it is just an example to show the main principles of how it works — then programmer could understand what is going on there, but not just use a useful pattern.

But yeah, I think to change it on “parent.prototype” instead of “parent” — in respect of DRY code reuse, it is better.

And regarding exactly code reuse I also like approach with using just “this._super()” to call the parent method with the same name. And it works for every method — in every child method “this._super()” is correctly set to needed method.

This approach is also known (for every child method with the same name a wrapper is needed; and in this wrapper correct value of “this._super” is set, then called parent method with the same name, and then “this._super” is restored), although it is less efficient by performance.

Also it is possible to implement “this._super” more efficiently if to use non-standard *caller* property and to store the *name* property for every added method. Then we can in calling “this._super()” to get correct parent context (via caller) and to call the method with the same name (having *name* property stored in function) from the parent prototype.

Unfortunately, *arguments.caller* is deprecated in strict mode of the ECMA-262-5 (although, I don’t like strict mode itself). However, if you use your own system and sure in it, then you can use it on full force and this approach is very good for code reuse, at least much better than every time to repeat “ChildName.superclass.” as it is in many frameworks (e.g. in ExtJS). Much better to have:

```
function doThat(arg) {
  // instead of
  // Child.superclass.doThat.call(this, arg)
  // we just do
```

```
    this._super(arg);
    // other actions
  }
```

Dmitry.

[#permalink](#)

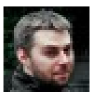
9. John Merge
8. April 2010 at 13:14

Nice!

BTW, why do you dislike strict mode of the ECMA-262-5?

John

[#permalink](#)

10.  [Dmitry A. Soshnikov](#)
8. April 2010 at 18:18

@**John Merge**

> why do you dislike strict mode of the ECMA-262-5

If to be more exact, not a strict mode itself, but the *splitting* on strict and non-strict. Because exactly this fact (of splitting) may cause many useless holywars and debates on forums. Some will tell that professional code is only in strict, but it is not so.

I of course understand reasons for which this feature is provided. The general is the deprecation of some features. Although, I have no idea how this voting was made — what to exclude and what do not.

In this way Python did the migration from 2.* to 3.* version in more radical manner. They just stop to support the old stylistics and syntax constructs making some of them 3.* completely incompatible with 2.* (although, I think that before this migration there was also some notifications about deprecated things).

But the thing is that even after the migration on new stylistics and syntax constructs, “use strict” can continue to exists as backward compatibility (as it is in Perl — ask any professional Perl programmer whether he uses strict mode – he’ll answer: “Yes of course”, but ask then “why?” — there will be no unequivocal answer), providing useless overloading in code. And of course the most funny answer is “Programming in strict mode is more professional”.

So I am not against stylistics that used in strict mode, I just think that there is no much sense in splitting on strict and non-strict.

I myself can easily use both stylistics including strict. That’s completely OK for me.

But what I really don’t like in ES5 — the stylistics and new approach of Object’s methods. I again understand reasons for which the did so. It is sort of insurance from that user can simply overwrite e.g. method *keys* if it would be defined in *Object.prototype*.

So they've decided to put these methods directly in *Object* constructor, but not in *Object.prototype*. What will again cause *procedural* stylistics instead of *object-oriented*. Moreover, there is already different stylistist for e.g. *hasOwnProperty* which is defined in *Object.prototype* — and the different stylistics for the same semantically entities (both are methods related to objects) is not so good approach.

Having in ES5 control for *[[Enumerable]]* (and other) internal property (in ES3 it is *{DontEnum}*), in own project in which you are sure and completely trust (and therefore can use the language for full force without any limitations) the first thing which can be done, is defining the same methods in *Object.prototype* with *{enumerable: false}* descriptor.

Compare e.g. this (abstract) example:

```
var o = Object.freeze(
  Object.seel(
    Object.defineProperties(
      Object.create(proto),
      properties
    )
  )
);
```

with this one:

```
var o = Object.create(proto)
  .defineProperties(properties)
  .seel()
  .freeze();
```

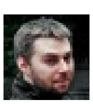
Obviously this "Object." prefix which we should repeat every time is not so good for code reuse and the whole stylistics in general looks worth in first case.

But all this of course is just my own opinion 😉

I wrote about it before:
http://groups.google.ru/group/comp.lang.javascript/msg/7e6f16761ef5c38f?hl=en

Dmitry.

#permalink

11.  Dmitry A. Soshnikov
27. April 2010 at 13:58

Thanks **Garrett Smith** for some English corrections.

Also, as addition for corresponding section of this article, you can read similar Garrett's article about property accessors — http://dhtmlkitchen.com/?category=/JavaScript/&date=2007/10/05/&entry=How-Property-Access-Works which puts to rest the myth of “everything in JS is an object”.

Dmitry.

12. gniavaj [#permalink](#)
27. September 2010 at 13:14

great article!!!

but i got a problem today,and i don’t know why.
can you explain for me

```
var istype = function(obj)
{
    debugger;
}
(function(){})(
    alert("i am running!");
);
```

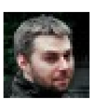
when the program run to the anonymous function on firefox, the istype function is called.

it confused me

13. gniavaj [#permalink](#)
27. September 2010 at 13:19

sorry,the anonymous function should be like this:

```
(function(){
    alert("i am running!")
})();
```

14.  [Dmitry A. Soshnikov](#) [#permalink](#)
27. September 2010 at 13:36

@gniavaj

> when the program run to the anonymous function on firefox, the istype function is called.

It’s the subtle case of ASI (Automatic semicolon insertion) mechanism. The surrounding parentheses of your second immediately invoked function, actually the *call parentheses of the first function*.

I.e. the first anonymous function is created and also immediately executed (at this moment it isn't even assigned yet to `istype` variable). You may rewrite this code in this way:

```
// pass the second function as
// argument for the first one
(function(obj) {...}(function(){alert("i am running!")}));
```

Thus, as you see, the *second function is passed as an argument* for the first one.

Accordingly, if you *call the second function* before passing as argument, then the result of the second function is passed as the argument for the first one function:

```
// pass the result of the second function -
// see call parentheses - (), the result is
// undefined, and this will be the value of
// "obj" argument
(function(obj) {...}(function(){alert("i am running!")()}));
```

And only after that the result of the first function is assigned to the `istype` variable (which is also `undefined` — i.e. the implicit returned value of the first function).

To fix your situation, just put explicit semicolon after the first function. Thus, the parser understands where the first part ends and starts the second one:

```
var istype = function(obj)
{
    debugger;
};
(function(){
    alert("i am running!")
})();
```

P.S.: Take a look also on [Chapter 5. Functions](#).

Dmitry.

15. insector [#permalink](#)
1. December 2010 at 01:35

Awesome articles, Dmitry. Been a professional JavaScript developer for eleven years and this still gave quite a bit of insight into a few things that I was unaware of.

Someone should have told people about JavaScript OO internals ten years ago, I salute you for making this effort!

16. Richard Durr [#permalink](#)
22. December 2010 at 16:14

A very informative and well written article. Thanks!

Would have saved me two weeks of recherches, if I'd found it earlier 😃

[#permalink](#)

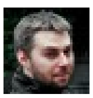
17. Struppi
23. February 2011 at 20:13

> (as it is in Perl — ask any professional Perl programmer whether he uses strict mode – he'll answer: "Yes of course", but ask then "why?" — there will be no unequivocal answer), providing useless overloading in code. And of course the most funny answer is "Programming in strict mode is more professional".

No, the only answer is, that you avoid typos – that's the only, but good reason.

But I think there are some difference between use strict in Perl and JS. But I won't argue for strict in JS, because I don't know enough about it.

[#permalink](#)


18. [Dmitry A. Soshnikov](#)
23. February 2011 at 20:35

@**Struppi**

Oh, it was my previous thoughts about strict mode (and this comment above was written before I wrote a detail analysis on strict mode and dug it deeply).

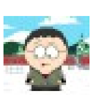
Now my meaning is changed since strict mode in ES5 (and in Perl I guess, though I'm not a Perl programmer) is a *transitional version* of the language. The next version, ES6 will be based exactly on ES5-strict.

The thing I mentioned (which probably I don't like) is exactly *splitting* the language on strict and non-strict. I.e. *constant* presence of these two modes. If to accept that this mode is only transitional (and in ES6 we won't have to choose the mode) it's completely OK — just a graceful transition from old version (with deprecated stuff) to the new one.

A detailed info on strict mode exactly in ES can be found in the appropriate [ECMA-262-5 in detail. Chapter 2. Strict Mode.](#).

Dmitry.

[#permalink](#)


19. Senxiv
10. March 2011 at 10:30

Dmitry,
I don't quite understand the inheritance section. Since you can get 'x' via `A.prototype` anyway, why is it so important to make it native? The second approach is advanced, but it's complicated. I

read it twice to figure it out.

In java(static classical language), inheritance is intuitive, you can understand the relationships among classes, instances at first glance. But with javascript, all the prototype properties, __proto__, constructor properties, it's very complicated. (Oh, thank you for your figure 3 in JavaScript: the core article. It helps a lot understanding prototype chain.)

Are there any best practices in JavaScript inheritance? Please suggest some reading materials. 🙂

#permalink

20.  Dmitry A. Soshnikov
10. March 2011 at 15:43

@**Senxiv**

> Since you can get 'x' via `A.prototype` anyway, why is it so important to make it native?

It's only to imitate the approach with classes — there state variables are native, not inherited. If you use a prototypical approach you may not create own `x`, but reuse it from the parent prototype.

Notice though, that there is a subtle case with object properties here. E.g.:

```
// constructor

function Foo(name) {
  this.name = name;
}

// prototype (shared) properties

Foo.prototype.data = [1, 2, 3];

Foo.prototype.showData = function () {
  console.log(this.name, this.data);
};

// instances

var foo1 = new Foo("foo1");
var foo2 = new Foo("foo2");

// both instances use
// the same default value of data

foo1.showData(); // "foo1", [1, 2, 3]
foo2.showData(); // "foo2", [1, 2, 3]

// however, if we change the
// data from one instance

```

```
foo1.data.push(4);

// it mirrors on the second instance

foo1.showData(); // "foo1", [1, 2, 3, 4]
foo2.showData(); // "foo2", [1, 2, 3, 4]
```

So in case when we need own properties (i.e. per instance), we create them in the constructor, not on the prototype. The prototype though can store some default values, but the case above should be considered.

> In java(static classical language), inheritance is intuitive, you can understand the relationships among classes, instances at first glance. But with javascript, all the prototype properties, __proto__, constructor properties, it's very complicated.

That's why you may create such a wrapper and program in classical approach not bothering with prototypal nature. But actually, there is no big difference in classical approach and prototypal — in both cases the inheritance chain is considered: in the class-based system it's a chain of classes, in the prototype-based — it's a prototype-chain.

Take a look only on CoffeeScript's classes: http://jashkenas.github.com/coffee-script/#classes You may see how Coffee compiles its code into JS and to understand how JS works.

> Are there any best practices in JavaScript inheritance? Please suggest some reading materials.

It depends on the situation. In one case it can be convenient to program in classical approach. In other one — to use the prototypal one. You may read also chapter 1 of the ES5 series where the OOP API of ES5 (with controlling property attributes, with inheriting without constructors via `Object.create`, etc) is described.

Dmitry.

21. Jiang #permalink
30. April 2012 at 00:52

For the "Feature of instanceof operator" part, you said that "All the instanceof operator does is only takes an object prototype — foo.[[Prototype]], and checks its presence in the prototype chain, starring the analysis from the Foo.prototype."

I have tried a demo as below:

```
function B() {}
var b = new B();
alert(b instanceof B); // true

var c1 = {};
var c2 = {};
var c3 = {};
c1.prototype = c2;
```

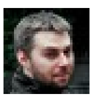
```
c2.prototype = c3;
c3.prototype = B.prototype;
alert(b instanceof c3); // true
```

But it says that "Uncaught TypeError: Expecting a function in instanceof check, but got #<B>" under Chrome.
After I changed the code as below:

```
function B() {}
var b = new B();
alert(b instanceof B); // true

var c1 = {};
var c2 = {};
var c3 = function() {};
c1.prototype = c2;
c2.prototype = c3;
c3.prototype = B.prototype;
alert(b instanceof c3); // true
```

So, I think maybe the function check is first applied for the instanceof operator, after that, just as you have mentioned in this article.

[#permalink](#)

22.  [Dmitry Soshnikov](#)
3. May 2012 at 19:34

@Jiang

Yes, absolutely correct, there is such a check first. Though this should go without saying, because the name of the `instanceof` operator already assumes that it works with an instance on the left hand side and with the *constructor function* on the right hand side.

Notice also, that in first your example `c1.prototype = c2;` does nothing special but just creates a casual property `prototype` on the object, it doesn't setup inheritance in this case, since again `c1` is not a constructor. If you want to play with inheritance of simple object, try using `c1.__proto__ = c2;` (works not in all browsers), or from ES5 — `var c1 = Object.create(c2);`.

[#permalink](#)

23. Jiang
11. June 2012 at 04:32

Thanks your pointing out. yes as you said, my demo does not setup the inheritance by setting the prototype property.

[#permalink](#)

24. Honza Joska
16. August 2012 at 06:12

I bow down to you mister. It took me 4 days to fully understand things presented in your marvelous articles and mainly this one.

JSFiddle site helped me alot with the testing

#permalink

25. Houde
10. September 2012 at 19:25

Hi this is amazing series of article. But I still did get clear about the Function and prototype:

```
function A() {}
function B() {}
A.prototype = B.prototype;
A.prototype.constructor === B.prototype.constructor //true
new A() === new B() //false? why?
A() === B() //true
A() === new A() //false? why?
```

Per my understanding, `new function()` will invoke the constructor of function, `A` and `B` have the same constructor, but `new A() === new B()` returned `false`. Can you show the essential analysis of this evalution?

#permalink

26. piglite
25. September 2012 at 21:16

even new A()===new A() is false:) LOL~
because, in my opinion, "===" depend on the memory location.everytime you call the constructor function, that will creat a new instance. and the new instance means in the different memory location.

#permalink

27. bird
28. September 2012 at 03:48

```
1.toString(); // SyntaxError!
```

Hi,I wonder what this error throw from ? Could you explain that for me.Thx!

#permalink

28. bird
6. October 2012 at 23:45

// Algorithm of evaluating a.test = 100:

1. wrapper = new Number(a);

2. wrapper.test = 100;
3. delete wrapper;

Why has the third step.. In es5, [[PUT]] internal method didnt say anything for the temp object to delete?

[#permalink](#)

29.  [Dmitry Soshnikov](#)
7. October 2012 at 12:17

@**bird**

> ```
> 1.toString(); // SyntaxError!
> ```
>
> Hi,I wonder what this error throw from ? Could you explain that for me.Thx!

The dot (point) is treated as the separator of the float part, since you can write it in short notation:

```
1. // 1.0
.5 // 05
1. + .5 // 1.5
```

The second dot is already property accessor, therefore:

```
1..toString();

// or with a space
1 .toString();

// or with parens
(1).toString();

// or brackets notation
1['toString']()
```

> // Algorithm of evaluating a.test = 100:
>
> 1. wrapper = new Number(a);
> 2. wrapper.test = 100;
> 3. delete wrapper;
>
> Why has the third step.. In es5, [[PUT]] internal method didnt say anything for the temp object to delete?

It can be deleted by GC since there is no any other reference to this intermediate wrapper object.

[#permalink](#)

30. bird
22. October 2012 at 07:33

@dmitry
thanks very much !!!

31. piglite [#permalink](#)
19. November 2012 at 00:59

@Dmitry Soshnikov

Hi,bother you again 🙂 but there is still a question need your help~ thx a lot~
As you write the code at the paragraph about the type conversion:

```
var a = {
  valueOf: function () {
    return 100;
  },
  toString: function () {
    return '__test';
  }
};

// in this operation
// toString method is
// called automatically
alert(a); // "  test"
```

The last statement, `alert(a);`, is it same as `alert(a.[[DefaultValue]](String))`? Use String as the parameter "hint"'s type.

32.  [Dmitry Soshnikov](#) [#permalink](#)
19. November 2012 at 21:59

@piglite

Yes, correct. Notice though, that `[[DefaultValue]]` is called only for objects, as in this example (as the result of `ToPrimitive` operation).

33. piglite [#permalink](#)
20. November 2012 at 00:43

@Dmitry Soshnikov

Thanks a lot！ you know, sometimes read the ES manual is a totally hard working, and really need a mentor just like you!
And this time, again, I need your help!
When I read the part of "Constructor" and "Prototype" especially the code in 2.3.4, I got a

inference, but I am not sure it is correct or not: Any function has two built-in properties “prototype” and “prototype.constructor”, and the value of “prototype.constructor” is the function itself just the time when it was used as constructor.

34. Dennie [#permalink](#)
21. November 2012 at 03:10

Wow. What a great article! This is for me so far the clearest most comprehensive article I have read on Javascripts prototype.
It helped me a lot!! Thanks

35.  [Dmitry Soshnikov](#) [#permalink](#)
21. November 2012 at 14:35

@**Dennie**, thanks, glad it’s useful.

@**piglite**, yeah, that’s correct. The `prototype` property is created for every function, and this property contains the `constructor` property which refers back to the function.

36. spyke [#permalink](#)
8. July 2013 at 13:43

I have a problem with property superproto …. my browser does not recognize

in the function

```
function B(){
    // o simplemente A.apply(this, arguments)
    B.superproto.constructor.apply(this, arguments);
}
```

37. A.K. [#permalink](#)
28. November 2013 at 04:58

Thanks a lot for this set of write-ups. I am an experienced C++/ruby programmer but have been struggling with trying to understand prototypes. I read more than 5 different descriptions (including at the MDN site and Crockford book) but this is the first time that I have really understood prototype linkage in JavaScript. It was getting so frustrating that I was planning to open up the source code of a JS implementation to see how prototype is implemented. Thanks for saving me the trouble.

38.  Hong [#permalink](#)
29. July 2014 at 08:11

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

I have one question about inheritance section. Please help.

Why we need one empty intermediate constructor `F`? just as following `Form1` code segment. It seems `Form2` can get the same result. Whether this `Form2` is OK?

In addition, is `superproto` a **user-defined** property? seems I cannot find it in ECMA-262-3 Spec.

Form1

```
function A() {
  console.log('A.[[Call]] activated');
  this.x = 10;
}
A.prototype.y = 20;
var a = new A();
console.log([a.x,a.y]);

function B() {
  B.superproto.constructor.apply(this, arguments);
}

var F = function () {};
F.prototype = A.prototype;
B.prototype = new F();
B.superproto = A.prototype;
B.prototype.constructor = B;
var b = new B();
console.log([b.x,b.y]);
```

Form2

```
function A() {
  console.log('A.[[Call]] activated');
  this.x = 10;
}
A.prototype.y = 20;
var a = new A();
console.log([a.x,a.y]);

function B() {
  A.apply(this, arguments);
}

B.prototype = A.prototype;
B.prototype.constructor = B;
var b = new B();
console.log([b.x,b.y]);
```

Regards.

[#permalink](#)

39.  Hong
26. November 2014 at 06:21

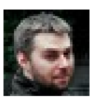
@**Dmitry**, could you please help me about above question? I still cannot understand the exact reason why we need one intermediate function `F`.

Also you mentioned that

> Notice how we create own property x on b instance: we call parent constructor via the B.superproto.constructor reference in context of newly created object.
>
> We have fixed also the issue with non-needed call of the parent constructor for creating the descendant prototype. Now the message "A.[[Call]] activated" is shown when is needed.

Because `B.superproto.constructor`, parent constructor will be called each time when creating descendant instance, so `A.[[Call]] activated` will also printed each time, which meaning the issue still exists, is it right?

[#permalink](#)

40.  [Dmitry Soshnikov](#)
29. November 2014 at 10:09

@**Hong** the difference in your Form2, is that after this:

```
B.prototype = A.prototype;
B.prototype.constructor = B;
```

Any modification to `B.prototype` (e.g. adding new method) will be reflected on instances of the constructor `A`.

```
B.prototype = A.prototype;
B.prototype.constructor = B;
...
B.prototype.foo = function() { console.log('foo'); };

// create an instance of A
var a = new A();
a.foo(); // what? why `a` has `foo` method?
console.log(a.constructor); // B? why B? it should be A.
```

The `superproto` is just user-level property described in this article. Instead of `superproto`, in many libraries people do `superclass` instead:

```
B.superclass = A;

// call super method from B method:
```

```
B.superclass.prototype.parentMethod.call(this);
```

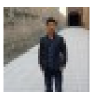
With `superproto` could be:

```
B.superproto = A.prototype;

// call super method from B method:
B.superproto.parentMethod.call(this);
```

> so A.[[Call]] activated will also printed each time, which meaning the issue still exists, is it right?

No, the issue doesn't exist — in this case we *want* to call the `A` constructor in context of new `B` instance. The issue was when we call `new A()` to initialize the `B.prototype`. With intermediate function `F` it's not called at that time.

Dmitry

41.  Hong [#permalink](#)
1. December 2014 at 05:10

@Dmitry.

Thanks for your help. I finally get it.

`F` and `intermediate Obj` created by `new F();` can be considered to be one intermediate isolation layer between parent and descendant children. We can take it as one abstract interface.

I think I have messed up `B.prototype` and `intermediate Obj` created by `new F();`. I was previously trapped into the confusion `B.prototype` is exactly the one of that `intermediate Obj`. While based on the evaluation strategy, `intermediate Obj` should be passed by sharing. `B.prototype` should be the address copy of that `intermediate Obj`, rather than the `intermediate Obj` itself. That is the reason why I think `Form1` and `Form2` are same.

Actually `B.prototype = new F();` should be the same form as following, right?

```
var interObj = new F();
B.prototype = interObj;
delete interObj;
```

For the issue, I think I misunderstand your meaning, issue you mentioned in the article is `non-needed call of the parent constructor for creating the descendant prototype`, while my mentioned issue is when using the new method, `new B();` will still call the parent constructor, because `B.superproto.constructor.apply(this, arguments);`.

In addition, another doubt point.

When we change the prototype of one function, we always update the constuctor property in the new prototype to the function itself at the same time.

Actually, I also do this step each time. While I am not very clear about the actually meaning of this step. Since the constructor is actually meaningless after creation of its objects. So whether the constructor is the function itself or other ones is really one important point? Please give some comment. Thanks.

Hong

[#permalink](#)

42.  [Dmitry Soshnikov](#)
1. December 2014 at 16:44

@**Hong**

Yes, your understanding is correct now.

> Since the constructor is actually meaningless after creation of its objects. So whether the constructor is the function itself or other ones is really one important point?

Well, it's theoretically meaningless. In practice one can want to check (if nothing was changed!) by which constructor an instance is created by checking its `constructor` property:

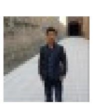
```
var a = new A;
var b = new B;

function foo(instance) {
  if (instance.constructor === A) {
    console.log('A instance');
  } else if (instance.constructor === B) {
    console.log('B instance');
  } else {
    ...
  }
}

foo(a); // 'A instance'
foo(b); // 'B instance'
```

In practice also often the `instanceof` check is used instead (although, it doesn't check the `constructor` property for this, but analyzes prototype chain):

```
if (instance instanceof A) {
  ...
}
```

Dmitry

[#permalink](#)

43.  Hong
3. December 2014 at 08:15

@Dmitry

Thanks, I get it now.

Thanks again for your series of excellent articles. I get a lot.

I find I can answer other guys' questions now 🙂

Hong

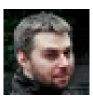
44.  Hong
3. December 2014 at 21:05 [#permalink]

@Dmitry

I remember you once mentioned you want to write one book.

Have you finish it? I will buy one 🙂

Hong

45.  [Dmitry Soshnikov]
3. December 2014 at 21:14 [#permalink]

@**Hong**, no, not yet 🙂 But the format of the online blog is OK too I think. The articles can be updated, typos can be fixed, etc — it's hard to do in a static book. We'll see, maybe in the future. In any case, if you can spread this knowledge, it's already good 😉

46. Marcos
30. January 2015 at 04:53 [#permalink]

One of explicit ways to call ToObject is to use built in Object constructor as a function

```
var n = Object(1); // [object Number]
var s = Object('test'); // [object String]

// also for some types it is
// possible to call Object with new operator
var b = new Object(true); // [object Boolean]

// but applied without arguments,
// new Object creates a simple object
var o = new Object(); // [object Object]

// in case if argument for Object function
```

```
// is already object value,
// it simply returns
var a = [];
alert(a === new Object(a)); // true
alert(a === Object(a)); // true
```

if it was explicit you have to indicate it in the code but in the example(below of the cite) i don't see `ToObject` applied, therefore is not it implicit?

[#permalink]

47.  [Dmitry Soshnikov]
31. January 2015 at 21:38

@**Marcos**, yes, what is meant there, is that `ToObject` can be called implicitly in some intermediate results (e.g. access a property on a primitive value), but in case on calling `Object(...)` we have an intent to call `ToObject` explicitly (which is called underneath of the `Object(...)` call).

[#permalink]

48. Margarito
22. August 2015 at 16:24

Great post.

[#permalink]

49.  Lorenzo
21. March 2016 at 04:35

Hi Dmitry,

Thanks for your posts, really interesting and useful.

In chapter 3.1 'Property constructor', in the following code:

```
function A() {}
A.prototype = {
  x: 10
};

var a = new A();
alert(a.x); // 10
alert(a.constructor === A); // false!
```

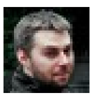
there is maybe a typo?
you were showing that if you change the prototype the reference is lost, so I think you need to assign a new object to `A.prototype`?

```
function A() {}
A.prototype = {
```

```
  x: 10
};

var a = new A();

alert(a.x); //10
alert(a.constructor === A.prototype.constructor); //true

A.prototype = {
 x: 20
};

alert(a.x); //10
alert(a.constructor === A.prototype.constructor); //false
```

thanks!

Lorenzo

50.  [Dmitry Soshnikov](#) [#permalink](#)
21. March 2016 at 14:17

@Lorenzo

> there is maybe a typo?
> you were showing that if you change the prototype the reference is lost, so I think you need to assign a new object to `A.prototype`?

No, there is no typo there. Yes, we assign a new object to `A.prototype`, aren't we (on line 2 in your first example)?

So when the instance `a` is created, it gets its `[[Prototype]]` set to `A.prototype`, which doesn't define the `constructor` property (so it's eventually found in the `Object.prototype`):

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

This also makes sense in your second example:

```
console.log(a.constructor === A.prototype.constructor); // true
```

However, as we said `A.prototype` by itself doesn't have own `constructor` property after our re-assignment to the `A.prototype`. So it's also found in the `Object.prototype` (which of course makes sense, since we assign a simple object `{ ... }` to `A.prototype`, and the constructor of this simple object is `Object`):

```
console.log(A.prototype.constructor === Object); // true
```

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