Dmitry Soshnikov in ECMAScript | 2010-09-02

JavaScript. The Core.

Read this article in: Japanese, German, Russian, French, Chinese, Polish.

> **Note:** a new 2nd Edition of this article is available.

This note is an overview and summary of the "ECMA-262-3 in detail" series. Every section contains references to the appropriate matching chapters so you can read them to get a deeper understanding.

Intended audience: experienced programmers, professionals.

We start out by considering the concept of an *object*, which is fundamental to ECMAScript.

An object

ECMAScript, being a highly-abstracted object-oriented language, deals with *objects*. There are also *primitives*, but they, when needed, are also converted to objects.

> An object is a *collection of properties* and has a *single prototype object*. The prototype may be either an object or the `null` value.

Let's take a basic example of an object. A prototype of an object is referenced by the internal `[[Prototype]]` property. However, in figures we will use `__<internal-property>__` underscore notation instead of the double brackets, particularly for the prototype object: `__proto__` .

For the code:

```
var foo = {
  x: 10,
  y: 20
};
```

we have the structure with two explicit *own* properties and one implicit `__proto__` property, which is the reference to the prototype of `foo` :

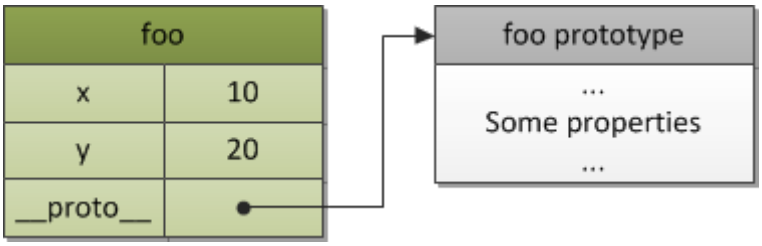

Figure 1. A basic object with a prototype.

What for these prototypes are needed? Let's consider a *prototype chain* concept to answer this question.

A prototype chain

Prototype objects are also just simple objects and may have their own prototypes. If a prototype has a non-null reference to its prototype, and so on, this is called the *prototype chain*.

> A prototype chain is a *finite* chain of objects which is used to implement *inheritance* and *shared properties*.

Consider the case when we have two objects which differ only in some small part and all the other part is the same for both objects. Obviously, for a good designed system, we would like to *reuse* that similar functionality/code without repeating it in every single object. In class-based systems, this *code reuse* stylistics is called the *class-based inheritance* — you put similar functionality into the class `A`, and provide classes `B` and `C` which inherit from `A` and have their own small additional changes.

ECMAScript has no concept of a class. However, a code reuse stylistics does not differ much (though, in some aspects it's even more flexible than class-based) and achieved via the *prototype chain*. This kind of inheritance is called a *delegation based inheritance* (or, closer to ECMAScript, a *prototype based inheritance*).

Similarly like in the example with classes `A`, `B` and `C`, in ECMAScript you create objects: `a`, `b`, and `c`. Thus, object `a` stores this common part of both `b` and `c` objects. And `b` and `c` store just their own additional properties or methods.

```
var a = {
  x: 10,
  calculate: function (z) {
    return this.x + this.y + z;
  }
};

var b = {
  y: 20,
  __proto__: a
};

var c = {
  y: 30,
  __proto__: a
};

// call the inherited method
b.calculate(30); // 60
c.calculate(40); // 80
```

Easy enough, isn't it? We see that `b` and `c` have access to the `calculate` method which is defined in `a` object. And this is achieved exactly via this prototype chain.

The rule is simple: if a property or a method is not found in the object itself (i.e. the object has no such an *own* property), then there is an attempt to find this property/method in the prototype chain. If the property is not found in the prototype, then a prototype of the prototype is considered, and so on, i.e. the whole

prototype chain (absolutely the same is made in class-based inheritance, when resolving an inherited *method* — there we go through the *class chain*). The first found property/method with the same name is used. Thus, a found property is called *inherited* property. If the property is not found after the whole prototype chain lookup, then `undefined` value is returned.

Notice, that `this` value in using an inherited method is set to the *original* object, but not to the (prototype) object in which the method is found. I.e. in the example above `this.y` is taken from `b` and `c`, but not from `a`. However, `this.x` is taken from `a`, and again via the *prototype chain* mechanism.

If a prototype is not specified for an object explicitly, then the default value for `__proto__` is taken — `Object.prototype`. Object `Object.prototype` itself also has a `__proto__`, which is the *final link* of a chain and is set to `null`.

The next figure shows the inheritance hierarchy of our `a`, `b` and `c` objects:

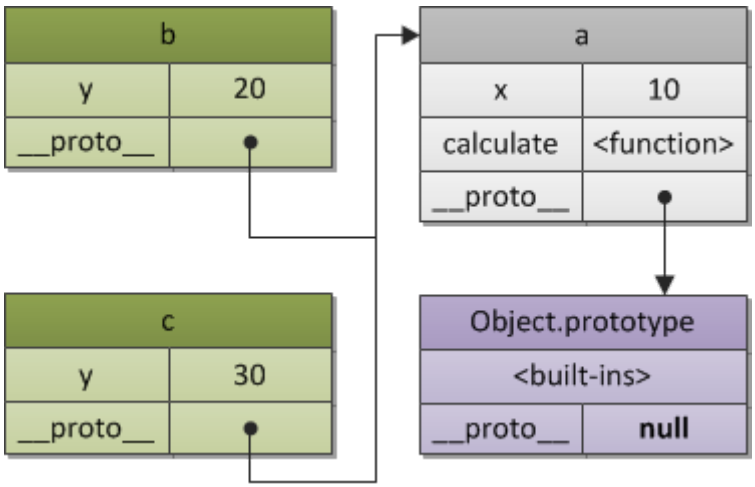

Figure 2. A prototype chain.

> Notice: ES5 standardized an alternative way for prototype-based inheritance using `Object.create` function:
>
> ```
> 1 var b = Object.create(a, {y: {value: 20}});
> 2 var c = Object.create(a, {y: {value: 30}});
> ```
>
> You can get more info on new ES5 APIs in the appropriate chapter.
>
> ES6 though standardizes the `__proto__`, and it can be used at initialization of objects.

Often it is needed to have objects with the *same or similar state structure* (i.e. the same set of properties), and with different *state values*. In this case we may use a *constructor function* which produces objects by *specified pattern*.

Constructor

Besides creation of objects by specified pattern, a *constructor* function does another useful thing — it *automatically sets a prototype object* for newly created objects. This prototype object is stored in the `ConstructorFunction.prototype` property.

E.g., we may rewrite previous example with `b` and `c` objects using a constructor function. Thus, the role of the object `a` (a prototype) `Foo.prototype` plays:

```
// a constructor function
function Foo(y) {
  // which may create objects
  // by specified pattern: they have after
  // creation own "y" property
  this.y = y;
}

// also "Foo.prototype" stores reference
// to the prototype of newly created objects,
// so we may use it to define shared/inherited
// properties or methods, so the same as in
// previous example we have:

// inherited property "x"
Foo.prototype.x = 10;

// and inherited method "calculate"
Foo.prototype.calculate = function (z) {
  return this.x + this.y + z;
};

// now create our "b" and "c"
// objects using "pattern" Foo
var b = new Foo(20);
var c = new Foo(30);

// call the inherited method
b.calculate(30); // 60
c.calculate(40); // 80

// let's show that we reference
// properties we expect

console.log(

  b.__proto__ === Foo.prototype, // true
  c.__proto__ === Foo.prototype, // true

  // also "Foo.prototype" automatically creates
  // a special property "constructor", which is a
  // reference to the constructor function itself;
  // instances "b" and "c" may found it via
  // delegation and use to check their constructor

```

```
46    b.constructor === Foo, // true
47    c.constructor === Foo, // true
48    Foo.prototype.constructor === Foo, // true
49
50    b.calculate === b.__proto__.calculate, // true
51    b.__proto__.calculate === Foo.prototype.calculate // true
52
53  );
```

This code may be presented as the following relationship:

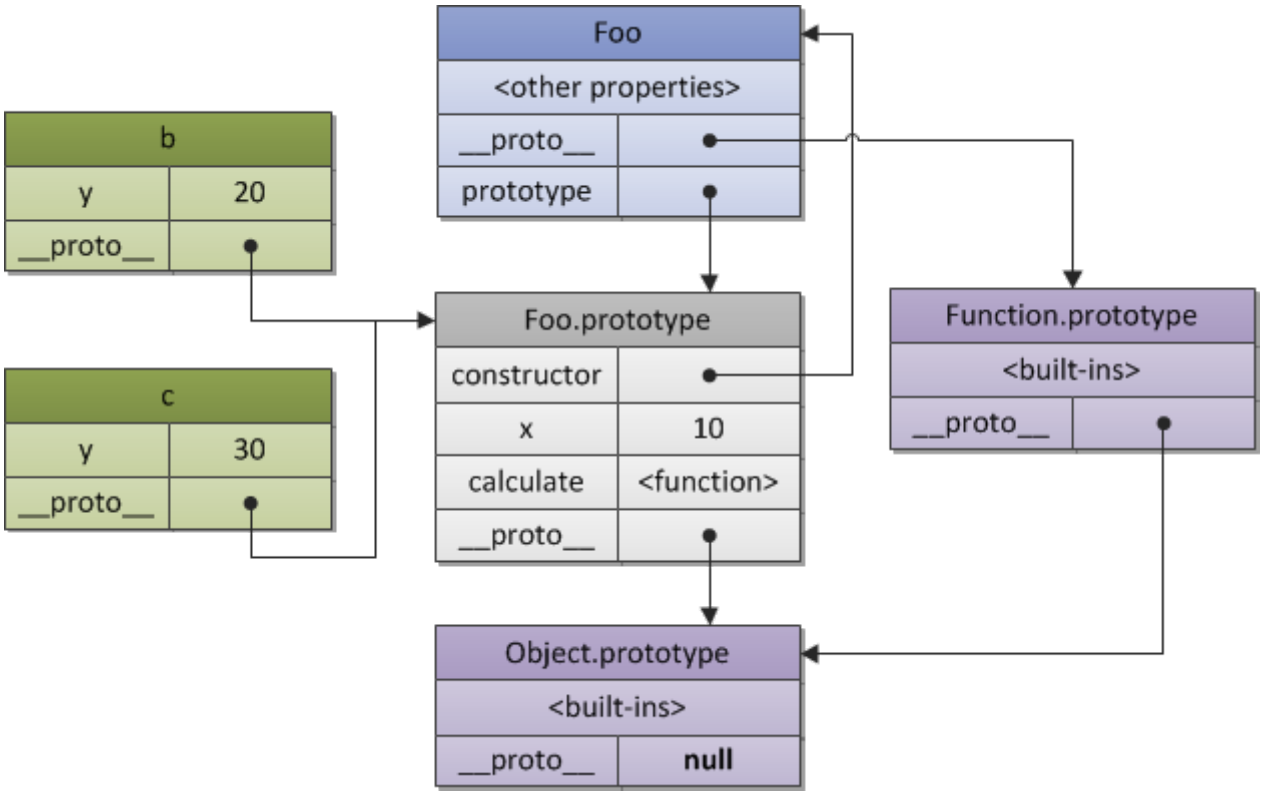

Figure 3. A constructor and objects relationship.

This figure again shows that every object has a prototype. Constructor function `Foo` also has its own `__proto__` which is `Function.prototype`, and which in turn also references via its `__proto__` property again to the `Object.prototype`. Thus, repeat, `Foo.prototype` is just an explicit property of `Foo` which refers to the prototype of `b` and `c` objects.

Formally, if to consider a concept of a *classification* (and we've exactly just now *classified* the new separated thing — `Foo`), a combination of the constructor function and the prototype object may be called as a "class". Actually, e.g. Python's *first-class* dynamic classes have absolutely the same implementation of properties/methods resolution. From this viewpoint, classes of Python are just a syntactic sugar for delegation based inheritance used in ECMAScript.

> Notice: in ES6 the concept of a “class” is standardized, and is implemented as exactly a syntactic sugar on top of 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
```

The complete and detailed explanation of this topic may be found in the Chapter 7 of ES3 series. There are two parts: Chapter 7.1. OOP. The general theory, where you will find description of various OOP paradigms and stylistics and also their comparison with ECMAScript, and Chapter 7.2. OOP. ECMAScript implementation, devoted exactly to OOP in ECMAScript.

Now, when we know basic object aspects, let’s see on how the *runtime program execution* is implemented in ECMAScript. This is what is called an *execution context stack*, every element of which is abstractly may be represented as also an object. Yes, ECMAScript almost everywhere operates with concept of an object 😉

Execution context stack

There are three types of ECMAScript code: *global* code, *function* code and *eval* code. Every code is evaluated in its *execution context*. There is only one global context and may be many instances of function and *eval* execution contexts. Every call of a function, enters the function execution context and evaluates the function code type. Every call of `eval` function, enters the *eval* execution context and evaluates its code.

Notice, that one function may generate infinite set of contexts, because every call to a function (even if the function calls itself recursively) produces a new context with a new *context state*:

```
function foo(bar) {}

// call the same function,
// generate three different
// contexts in each call, with
// different context state (e.g. value
// of the "bar" argument)

foo(10);
foo(20);
foo(30);
```

An execution context may activate another context, e.g. a function calls another function (or the global context calls a global function), and so on. Logically, this is implemented as a stack, which is called the *execution context stack*.

A context which activates another context is called a *caller*. A context is being activated is called a *callee*. A callee at the same time may be a caller of some other callee (e.g. a function called from the global context, calls then some inner function).

When a caller activates (calls) a callee, the caller suspends its execution and passes the control flow to the callee. The callee is pushed onto the the stack and is becoming a *running (active)* execution context. After the callee's context ends, it returns control to the caller, and the evaluation of the caller's context proceeds (it may activate then other contexts) till the its end, and so on. A callee may simply *return* or exit with an *exception*. A thrown but not caught exception may exit (pop from the stack) one or more contexts.

I.e. all the ECMAScript *program runtime* is presented as the *execution context (EC) stack*, where *top* of this stack is an *active* context:

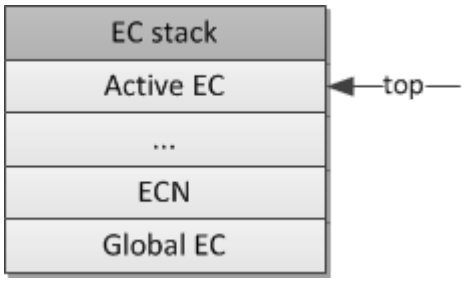

Figure 4. An execution context stack.

When program begins it enters the *global execution context*, which is the *bottom* and the *first* element of the stack. Then the global code provides some initialization, creates needed objects and functions. During the execution of the global context, its code may activate some other (already created) function, which will enter their execution contexts, pushing new elements onto the stack, and so on. After the initialization is done, the runtime system is waiting for some *event* (e.g. user's mouse click) which will activate some function and which will enter a new execution context.

In the next figure, having some function context as `EC1` and the global context as `Global EC`, we have the following stack modification on entering and exiting `EC1` from the global context:

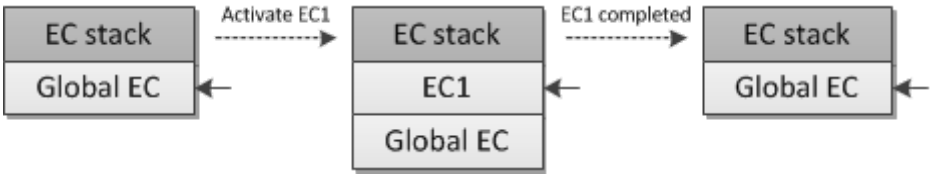

Figure 5. An execution context stack changes.

This is exactly how the runtime system of ECMAScript manages the execution of a code.

More information on execution context in ECMAScript may be found in the appropriate Chapter 1. Execution context.

As we said, every execution context in the stack may be presented as an object. Let's see on its structure and what kind of *state* (which properties) a context is needed to execute its code.

Execution context

An execution context abstractly may be represented as a simple object. Every execution context has set of properties (which we may call a *context's state*) necessary to track the execution progress of its associated code. In the next figure a structure of a context is shown:

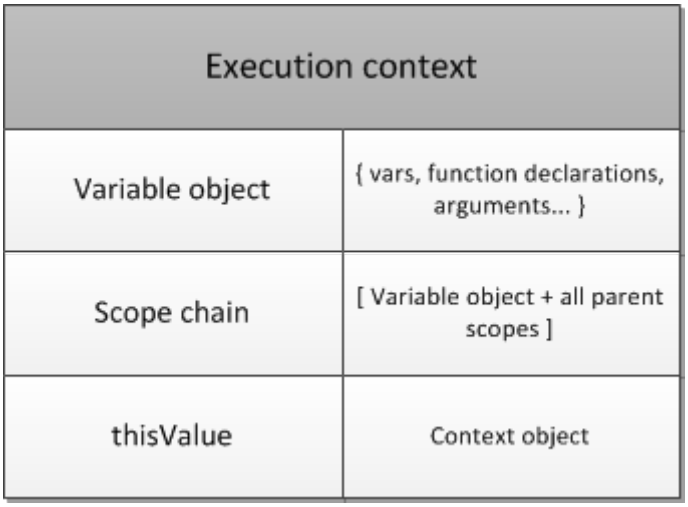

Figure 6. An execution context structure.

Besides these three needed properties (a *variable object*, a `this` *value* and a *scope chain*), an execution context may have any additional state depending on implementation.

Let's consider these important properties of a context in detail.

Variable object

> A *variable object* is a *container of data* associated with the execution context. It's a special object that stores *variables* and *function declarations* defined in the context.

Notice, that *function expressions* (in contrast with *function declarations*) are *not included* into the variable object.

A variable object is an abstract concept. In different context types, physically, it's presented using different object. For example, in the global context the variable object is the *global object itself* (that's why we have an ability to refer global variables via property names of the global object).

Let's consider the following example in the global execution context:

```
1  var foo = 10;
2
3  function bar() {} // function declaration, FD
4  (function baz() {}); // function expression, FE
5
6  console.log(
7    this.foo == foo, // true
8    window.bar == bar // true
```

```
);

console.log(baz); // ReferenceError, "baz" is not defined
```

Then the global context's variable object (VO) will have the following properties:

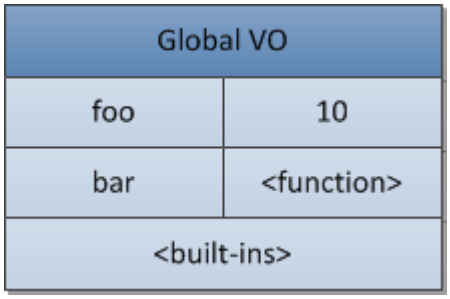

Figure 7. The global variable object.

See again, that function `baz` being a *function expression* is not included into the variable object. That's why we have a `ReferenceError` when trying to access it outside the function itself.

Notice, that in contrast with other languages (e.g. C/C++) in ECMAScript *only functions* create a new scope. Variables and inner functions defined within a scope of a function are not visible directly outside and do not pollute the global variable object.

Using `eval` we also enter a new (eval's) execution context. However, `eval` uses either global's variable object, or a variable object of the caller (e.g. a function from which `eval` is called).

And what about functions and their variable objects? In a function context, a variable object is presented as an *activation object*.

Activation object

When a function is *activated* (called) by the caller, a special object, called an *activation object* is created. It's filled with *formal parameters* and the special `arguments` object (which is a map of formal parameters but with index-properties). The *activation object* then is used as a *variable object* of the function context.

I.e. a function's variable object is the same simple variable object, but besides variables and function declarations, it also stores formal parameters and `arguments`

object and called the *activation object*.

Considering the following example:

```
function foo(x, y) {
  var z = 30;
  function bar() {} // FD
  (function baz() {}); // FE
}

foo(10, 20);
```

we have the next activation object (AO) of the `foo` function context:

Activation object	
x	10
y	20
arguments	{0: 10, 1: 20, ..}
z	30
bar	<function>

Figure 8. An activation object.

And again the *function expression* `baz` is not included into the variable/activate object.

The complete description with all subtle cases (such as *"hoisting"* of variables and function declarations) of the topic may be found in the same name Chapter 2. Variable object.

> Notice, in ES5 the concepts of *variable object*, and *activation object* are combined into the *lexical environments* model, which detailed description can be found in the appropriate chapter.

And we are moving forward to the next section. As is known, in ECMAScript we may use *inner functions* and in these inner functions we may refer to variables of *parent* functions or variables of the *global* context. As we named a variable object

as a *scope object* of the context, similarly to the discussed above prototype chain, there is so-called a *scope chain*.

Scope chain

> A *scope chain* is a *list of objects* that are searched for *identifiers* appear in the code of the context.

The rule is again simple and similar to a prototype chain: if a variable is not found in the own scope (in the own variable/activation object), its lookup proceeds in the parent's variable object, and so on.

Regarding contexts, identifiers are: *names* of variables, function declarations, formal parameters, etc. When a function refers in its code the identifier which is not a local variable (or a local function or a formal parameter), such variable is called a *free variable*. And to *search these free variables* exactly a *scope chain* is used.

In general case, a *scope chain* is a list of all those *parent variable objects*, *plus* (in the front of scope chain) the function's *own variable/activation object*. However, the scope chain may contain also any other object, e.g. objects dynamically added to the scope chain during the execution of the context — such as *with-objects* or special objects of *catch-clauses*.

When *resolving* (looking up) an identifier, the scope chain is searched starting from the activation object, and then (if the identifier isn't found in the own activation object) up to the top of the scope chain — repeat, the same just like with a prototype chain.

```
1  var x = 10;
2
3  (function foo() {
4    var y = 20;
5    (function bar() {
6      var z = 30;
7      // "x" and "y" are "free variables"
8      // and are found in the next (after
9      // bar's activation object) object
10     // of the bar's scope chain
11     console.log(x + y + z);
12   })();
13 })();
```

We may assume the linkage of the scope chain objects via the implicit `__parent__` property, which refers to the next object in the chain. This approach may be tested in a real Rhino code, and exactly this technique is used in ES5 *lexical environments* (there it's named an `outer` link). Another representation of a scope chain may be a simple array. Using a `__parent__` concept, we may represent the example above with the following figure (thus parent variable objects are saved in the `[[Scope]]` property of a function):

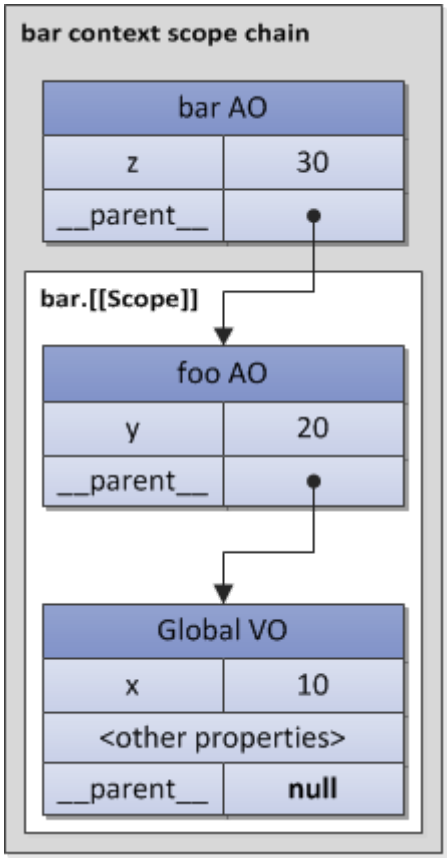

Figure 9. A scope chain.

At code execution, a scope chain may be augmented using `with` statement and `catch` clause objects. And since these objects are simple objects, they may have prototypes (and prototype chains). This fact leads to that scope chain lookup is *two-dimensional*: (1) first a scope chain link is considered, and then (2) on every scope chain's link — into the depth of the link's prototype chain (if the link of course has a prototype).

For this example:

```
Object.prototype.x = 10;

var w = 20;
var y = 30;
```

```
5
6  // in SpiderMonkey global object
7  // i.e. variable object of the global
8  // context inherits from "Object.prototype",
9  // so we may refer "not defined global
10 // variable x", which is found in
11 // the prototype chain
12
13 console.log(x); // 10
14
15 (function foo() {
16
17   // "foo" local variables
18   var w = 40;
19   var x = 100;
20
21   // "x" is found in the
22   // "Object.prototype", because
23   // {z: 50} inherits from it
24
25   with ({z: 50}) {
26     console.log(w, x, y , z); // 40, 10, 30, 50
27   }
28
29   // after "with" object is removed
30   // from the scope chain, "x" is
31   // again found in the AO of "foo" context;
32   // variable "w" is also local
33   console.log(x, w); // 100, 40
34
35   // and that's how we may refer
36   // shadowed global "w" variable in
37   // the browser host environment
38   console.log(window.w); // 20
39
40 })();
```

we have the following structure (that is, before we go to the `__parent__` link, first `__proto__` chain is considered):

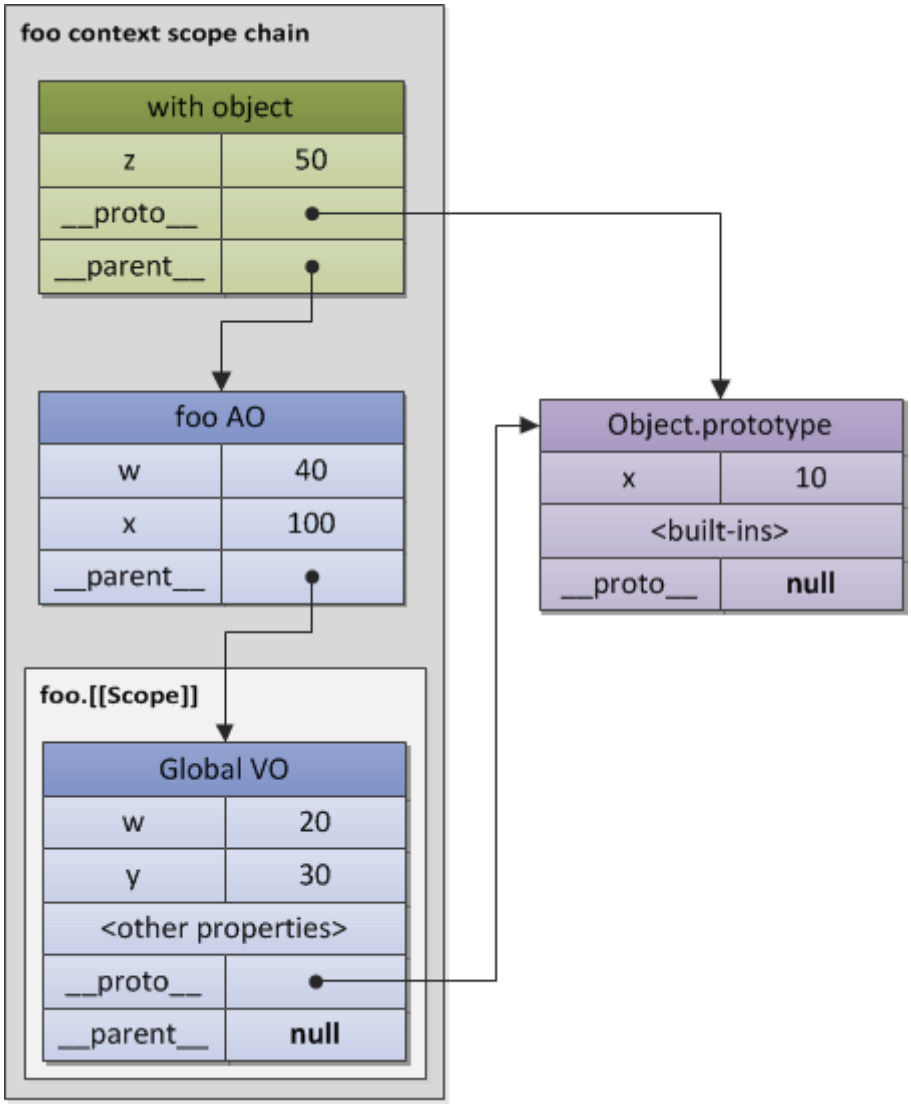

Figure 10. A "with-augmented" scope chain.

Notice, that not in all implementations the global object inherits from the `Object.prototype`. The behavior described on the figure (with referencing "non-defined" variable `x` from the global context) may be tested e.g. in SpiderMonkey.

Until all parent variable objects exist, there is nothing special in getting parent data from the inner function — we just traverse through the scope chain resolving (searching) needed variable. However, as we mentioned above, after a context ends, all its state and it itself are *destroyed*. At the same time an *inner function* may be *returned* from the parent function. Moreover, this returned function may be later activated from another context. What will be with such an activation if a context of some free variable is already "gone"? In the general theory, a concept which helps to solve this issue is called a *(lexical) closure*, which in ECMAScript is directly related with a *scope chain* concept.

Closures

In ECMAScript, functions are the *first-class* objects. This term means that functions may be passed as arguments to other functions (in such case they are called *"funargs"*, short from "functional arguments"). Functions which receive "funargs" are called *higher-order functions* or, closer to mathematics, *operators*. Also functions may be returned from other functions. Functions which return other functions are called *function valued* functions (or functions with *functional value*).

There are two conceptual problems related with "funargs" and "functional values". And these two sub-problems are generalized in one which is called a *"Funarg problem"* (or "A problem of a functional argument"). And exactly to solve the *complete "funarg problem"*, the concept of *closures* was invented. Let's describe in more detail these two sub-problems (we'll see that both of them are solved in ECMAScript using a mentioned on figures `[[Scope]]` property of a function).

First subtype of the "funarg problem" is an *"upward funarg problem"*. It appears when a function is returned "up" (to the outside) from another function and uses already mentioned above *free variables*. To be able access variables of the parent context *even after the parent context ends*, the inner function *at creation moment* saves in it's `[[Scope]]` property parent's *scope chain*. Then when the function is *activated*, the scope chain of its context is formed as combination of the activation object and this `[[Scope]]` property (actually, what we've just seen above on figures):

```
Scope chain = Activation object + [[Scope]]
```

Notice again the main thing — exactly at *creation moment* — a function saves *parent's* scope chain, because exactly this *saved scope chain* will be used for variables lookup then in further calls of the function.

```
function foo() {
  var x = 10;
  return function bar() {
    console.log(x);
  };
}

// "foo" returns also a function
// and this returned function uses
// free variable "x"

var returnedFunction = foo();

// global variable "x"
```

```
var x = 20;

// execution of the returned function
returnedFunction(); // 10, but not 20
```

This style of scope is called the *static (or lexical) scope*. We see that the variable `x` is found in the saved `[[Scope]]` of returned `bar` function. In general theory, there is also a *dynamic scope* when the variable `x` in the example above would be resolved as `20`, but not `10`. However, dynamic scope is not used in ECMAScript.

The second part of the "funarg problem" is a *"downward funarg problem"*. In this case a parent context may exist, but may be an ambiguity with resolving an identifier. The problem is: *from which scope* a value of an identifier should be used — statically saved at a function's creation or dynamically formed at execution (i.e. a scope of a *caller*)? To avoid this ambiguity and to form a closure, a *static scope* is decided to be used:

```
// global "x"
var x = 10;

// global function
function foo() {
  console.log(x);
}

(function (funArg) {

  // local "x"
  var x = 20;

  // there is no ambiguity,
  // because we use global "x",
  // which was statically saved in
  // [[Scope]] of the "foo" function,
  // but not the "x" of the caller's scope,
  // which activates the "funArg"

  funArg(); // 10, but not 20

})(foo); // pass "down" foo as a "funarg"
```

We may conclude that a *static scope* is an *obligatory requirement to have closures* in a language. However, some languages may provided combination of dynamic and static scopes, allowing a programmer to choose — what to closure and what do not. Since in ECMAScript only a static scope is used (i.e. we have solutions for both subtypes of the "funarg problem"), the conclusion is: *ECMAScript has complete support of closures*, which technically are implemented using `[[Scope]]` property of functions. Now we may give a correct definition of a closure:

> A *closure* is a combination of a code block (in ECMAScript this is a function) and statically/lexically saved all parent scopes. Thus, via these saved scopes a function may easily refer free variables.

Notice, that since *every* (normal) function saves `[[Scope]]` at creation, theoretically, *all functions* in ECMAScript *are closures*.

Another important thing to note, that several functions may have *the same parent scope* (it's quite a normal situation when e.g. we have two inner/global functions). In this case variables stored in the `[[Scope]]` property are *shared between all functions* having the same parent scope chain. Changes of variables made by one closure are *reflected* on reading these variables in another closure:

```
function baz() {
  var x = 1;
  return {
    foo: function foo() { return ++x; },
    bar: function bar() { return --x; }
  };
}

var closures = baz();

console.log(
  closures.foo(), // 2
  closures.bar()  // 1
);
```

This code may be illustrated with the following figure:

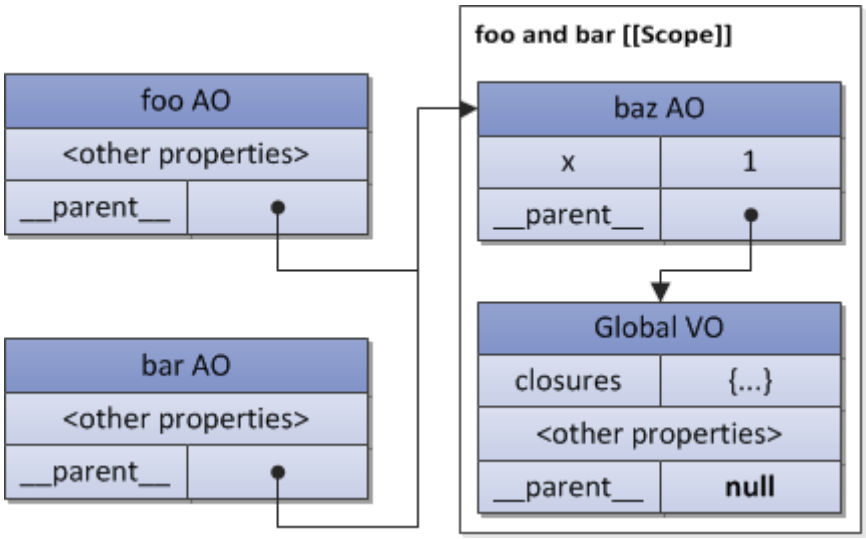

Figure 11. A shared [[Scope]].

Exactly with this feature confusion with creating several functions in a loop is related. Using a loop counter inside created functions, some programmers often get unexpected results when all functions have the *same* value of a counter inside a function. Now it should be clear why it is so — because all these functions have the same `[[Scope]]` where the loop counter has the last assigned value.

```
var data = [];

for (var k = 0; k < 3; k++) {
  data[k] = function () {
    console.log(k);
  };
}

data[0](); // 3, but not 0
data[1](); // 3, but not 1
data[2](); // 3, but not 2
```

There are several techniques which may solve this issue. One of the techniques is to provide an additional object in the scope chain — e.g. using additional function:

```
var data = [];

for (var k = 0; k < 3; k++) {
  data[k] = (function (x) {
    return function () {
      console.log(x);
    };
  })(k); // pass "k" value
}

// now it is correct
data[0](); // 0
data[1](); // 1
data[2](); // 2
```

NOTE: ES6 introduced *block-scope* bindings. This is done via `let` or `const` keywords. Example from above can now easily and conveniently rewritten as:

```
let data = [];

for (let k = 0; k < 3; k++) {
  data[k] = function () {
    console.log(k);
  };
}

data[0](); // 0
data[1](); // 1
data[2](); // 2
```

Those who interested deeper in theory of closures and their practical application, may find additional information in the Chapter 6. Closures. And to get more information about a scope chain, take a look on the same name Chapter 4. Scope chain.

And we're moving to the next section, considering the last property of an execution context. This is concept of a `this` value.

This value

> A `this` value is a special object which is related with the execution context. Therefore, it may be named as a *context object* (i.e. an object in which context the execution context is *activated*).

Any object can be used as `this` value of the context. One important note is that the `this` value is a *property of the execution context*, but *not* a property of the variable object.

This feature is very important, because in *contrast with variables*, `this` *value never participates in identifier resolution process*. I.e. when accessing `this` in a code, its value is taken *directly* from the execution context and *without any scope chain lookup*. The value of `this` is determined *only once*, on *entering the context*.

> NOTE: In ES6 `this` actually became a property of a *lexical environment*, i.e. property of the *variable object* in ES3 terminology. This is done to support *arrow functions*, which have *lexical* `this`, which they inherit from parent contexts.

By the way, in contrast with ECMAScript, e.g. Python has its `self` argument of methods as a simple variable which is resolved the same and may be even changed during the execution to another value. In ECMAScript it is *not possible* to assign a new value to `this`, because, repeat, it's not a variable and is not placed in the variable object.

In the global context, a `this value` is the *global object itself* (that means, `this` value here equals to *variable object*):

```
1
```

```
var x = 10;

console.log(
  x, // 10
  this.x, // 10
  window.x // 10
);
```

In case of a function context, `this` value in *every single function call* may be *different*. Here `this` value is provided by the *caller* via the *form of a call expression* (i.e. the way of how a function is activated). For example, the function `foo` below is a *callee*, being called from the global context, which is a *caller*. Let's see on the example, how for the same code of a function, `this` value in different calls (different ways of the function activation) is provided *differently* by the caller:

```
// the code of the "foo" function
// never changes, but the "this" value
// differs in every activation

function foo() {
  alert(this);
}

// caller activates "foo" (callee) and
// provides "this" for the callee

foo(); // global object
foo.prototype.constructor(); // foo.prototype

var bar = {
  baz: foo
};

bar.baz(); // bar

(bar.baz)(); // also bar
(bar.baz = bar.baz)(); // but here is global object
(bar.baz, bar.baz)(); // also global object
(false || bar.baz)(); // also global object

var otherFoo = bar.baz;
otherFoo(); // again global object
```

To consider deeply why (and that is more essential — *how*) `this` value may change in every function call, you may read Chapter 3. This where all mentioned above cases are discussed in detail.

Conclusion

At this step we finish this brief overview. Though, it turned out to not so "brief" 😉 However, the whole explanation of all these topics requires a complete book. We though didn't touch two major topics: *functions* (and the difference between some types of functions, e.g. *function declaration* and *function expression*) and the

evaluation strategy used in ECMAScript. Both topics may be found in the appropriate chapters of ES3 series: Chapter 5. Functions and Chapter 8. Evaluation strategy.

If you have comments, questions or additions, I'll be glad to discuss them in comments.

Good luck in studying ECMAScript!

Written by: Dmitry A. Soshnikov
Published on: 2010-09-02

Dmitry Soshnikov

Software engineer interested in learning and education. Sometimes blog on topics of programming languages theory, compilers, and ECMAScript.

Published

2010-09-02

Write a Comment

142 COMMENTS

← Older Comments

shadowhorst
2014-10-09

This is a great piece of work. Thank you very much for sharing this.

arch
2015-02-19

This is by far the best explanation that I have found on prototypal inheritance in JS. Thank you so much! greatly appreciated.

blajs
2015-04-12

Hi Dmitry, what is the difference between scope chain lookup and identifier resolution, are they same ? Thanks

Dmitry Soshnikov
2015-04-12

@**blajs**, yes, “scope chain lookup” and “identifier resolution” may be used interchangeably. The scope chain lookup is a *mechanism* which is used to resolve (i.e. to find) an identifier.

Tom
2015-06-03

Guy, thanks a lot for this excellent article! Eventually, I got the very special concepts of the prototype-based OO used in JavaScript, after so many years 🙂 Your samples are just awesome.

Aflext Yang

2015-08-08

Great Article! Thanks a lot!

Anand Sonake

2015-08-13

Fantastic article. This is something i was looking for.
GREAT JOB Dmitry.

manohar

2015-08-19

Fabulous

Sajid

2015-10-05

Thank You Dmitry,

Excellent series of articles.
It has clarified much of my misunderstanding.

Dmitry Soshnikov

2015-10-05

@**Sajid**, thanks, glad it’s useful.

J Park

2015-10-26

Thank you for much for this article. super good.

Andrej Sagaidak

2015-11-19

Very clearly explained. Thank you so much Dmitry

plusman

2015-12-06

Excellent !

magicdawn

2016-02-19

Great work!

the ES6(aka ES2015) brings new problem.

```
1  var data = [];
2  for (let k = 0; k < 3; k++) {
3    data[k] = function () {
4      alert(k);
5    };
6  }
7
8
9  data[0](); // 0
10 data[1](); // 1
11 data[2](); // 2
```

Does let change the scope chain? Does the k exist in the data_k's function VO scope ?

Dmitry Soshnikov
2016-02-20

@**magicdawn**,

> *Does let change the scope chain?*

Per ES6 `let` is a block-scoped. So every time we enter the block of the `for` loop, a new environment is created, and the function created inside the block, is created relatively to this environment.

Roughly it's similar to (although at engine level much more optimized than this):

```
var data = [];
for (var k = 0; k < 3; k++) {
  (function(x) {
    data[x] = function () {
      alert(x);
    };
  })(k);
}
```

magicdawn
2016-02-21

Thanks.

Faizal
2016-03-02

Awesome article.
Great!!

Balamurugan
2016-06-18

Thanks a lot Dmitry. Great work!!!.

I wish I found this earlier..

Much appreciated!!.

shyam
2016-07-04

Thanks Dmitry.
Great way of Explanation.
easy understandably.

Håkan MacLean
2016-07-12

THANK you Dmitry! After spending half a day reading about prototype and __proto__ this was the article that finally made me understand prototype inheritance in this weird language :).

Jayam Malviya
2016-09-21

For a newbie like me, it was a deep dive.
Thank you Dimtry, for such an amazing must read blog!!!

Petar
2016-10-09

Slava Rasija
Slava Javascript

Crni
2017-01-17

This article was helpful for me, and I clarified a lot. One thing bother me. When I executed code from Constructor segment of the article in Console of web browser, and check for `console.log(b.__proto__)` I got as result `Foo {x=10, calculate=function() }`. I thought result of `b.__proto__` should be `Foo.prototype` ?

Dmitry Soshnikov
2017-01-17

@**Crni**, yes, this is how Chrome's debug console represents `Foo.prototype`, i.e. an object containing properties `x`, and `calculate`, and `b.__proto__` points to it.

Crni
2017-01-18

Thanks on answer, but why it isn't named `Foo.prototype`, like it should be, but just `Foo`. It is in Chrome, Firefox, also. Is there some agreement not to call things as they should be called 🙂

Dmitry Soshnikov
2017-01-18

@**Crni**, that’s the question already to who built the debug consoles in Chrome, and Firefox 🙂 You can check the correct assumption with this equality:

```
1 | b.__proto__ === Foo.prototype; // true
```

Prakash saini
2017-03-12

This is so valuable information. I learned alot from it.
Thank you so much.

Prakash saini
2017-03-12

Hello sir,
Great article.
Can u please explain this below line.

```
1 | Foo.prototype.constructor === Foo, // true
```

Thanks

houcine
2017-03-16

Thanks a lot. I just finished this overview, and I’m hungry for more ECMA-262-3 in detail.

Dmitry Soshnikov
2017-03-16

@**houcine**, thanks, appreciated, and glad it's useful!

@**Prakash saini**, thanks for the feedback, appreciated.

> *Can u please explain this below line.*
>
> ```
> 1 Foo.prototype.constructor === Foo, // true
> ```

When a function is created, it receives the `prototype` explicit property, which is an object that also stores the `constructor` property, which in turns refers back to the function itself.

Something like:

```
1 function Foo() {}
2
3 // This happens implicitly:
4 let proto = {constructor: Foo};
5 Foo.prototype = proto;
6
7 console.log(Foo.prototype.constructor === Foo); // true
```

George
2017-05-04

Hi Dmitry,
An Object

```
1 var foo = {
2   x: 10,
3   y: 20
4 };
```

I understand the `foo.__proto__` equal `foo.constructor.prototype` or `Object.ptototype ,` not
equal `foo.prototype` ?

Dmitry Soshnikov
2017-05-05

@**George**,

If an object is created using literal notation `{}`, it's the same as `new Object()`. That is, its `constructor` property points to the `Object`.

```
const foo = {
  x: 10,
  y: 20,
};

console.log(
  foo.__proto__ === Object.prototype, // true
  Object.getPrototypeOf(foo) === Object.prototype, // true
  foo.constructor === Object, // true
  foo.constructor.prototype === Object.prototype, // true
  foo.prototype, // undefined
);
```

The explicit `prototype` property is set only for constructor functions. Created objects by this constructors, get their `__proto__` from `ConstructorFunction.prototype`. Since `foo` is not a constructor, there is `prototype` property on it.

Xin
2017-05-08

This is a nice article. Thanks for the work.

Vladan Andjelkovich
2017-07-19

Amazing explanation. This whole blog is amazing and valuable resource. Better that any Kyle Simpson's || Doug Crockford's master class!. Great job!!

Harmandeep Kaur
2017-08-08

Awesome Artcile! really helpful 🙂

Mithul Bhansali
2017-08-15

Great article on explanation of Javascript features.

Varrian
2017-10-26

Awesome! I have read several materials about closure. This one is the best one to explain all the concept step by step and finally put forward the closure.

ralph
2017-11-01

is execution context stack a continuous memories contains the value needed when execute the piece of the code?

It seems there is no way to inspect the execution context stack, there is no object corresponding to this stack, is it correct?

Dmitry Soshnikov
2017-11-01

@**ralph**, you can normally inspect an execution context stack (the call-stack), and the environment frames (activation objects) using a debugger. E.g. in Chrome:

```
function foo(x) {
  let y = 20;
  bar(y);
}

function bar(y) {
  let x = 10;

  // Here execution stops, and you
  // can introspect the stack.
  debugger;

  return z + y;
}
```

Cameron

2018-01-06

Fantastic article. It clarified so much for me. The diagrams especially helped me sort things out. Thanks!

aliaksei krauchanka

2018-01-08

Dmitry you’re the Lord of JS!
Thank you so much for all of this info about JS.

akhil

2018-01-09

I just couldnt understand the closure example,’x’ parameter ?

← Older Comments

Write a Comment

RELATED CONTENT BY TAG **[[SCOPE]]** **ACTIVATION OBJECT** **ECMA-262-3** **ECMASCRIPT** **EXECUTION CONTEXT** **JAVASCRIPT** **OOP** **SCOPE CHAIN** **THIS** **VARIABLE OBJECT**