Menu

•

ECMA-262

by Dmitry Soshnikov

JavaScript. The core.

Tweet Like 227 G+1 130 Share 104 Share

Read this article in: Japanese, German (version 2), Arabic, Russian, French, Chinese, Polish.

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`:

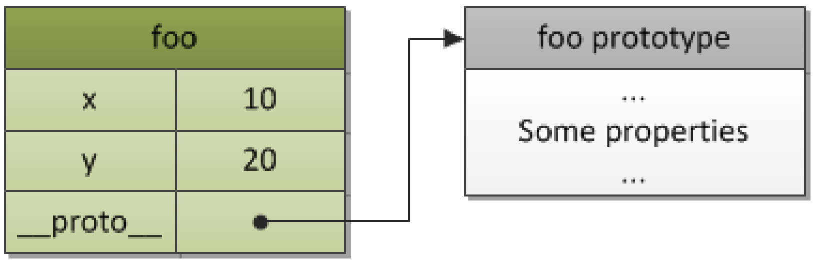


Figure 1. A basic object with a prototype.

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:

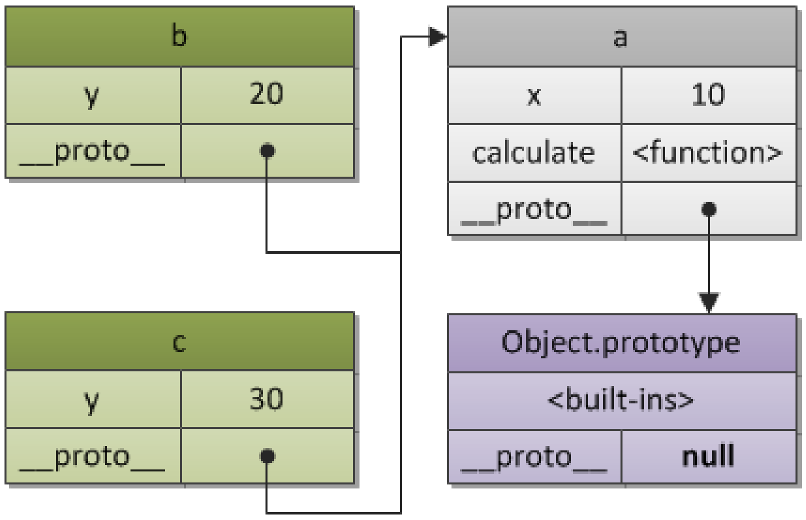


Figure 2. A prototype chain.

Notice: ES5 standardized an alternative way for prototype-based inheritance using `Object.create` function:

```
var b = Object.create(a, {y: {value: 20}});
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

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

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

);
```

This code may be presented as the following relationship:

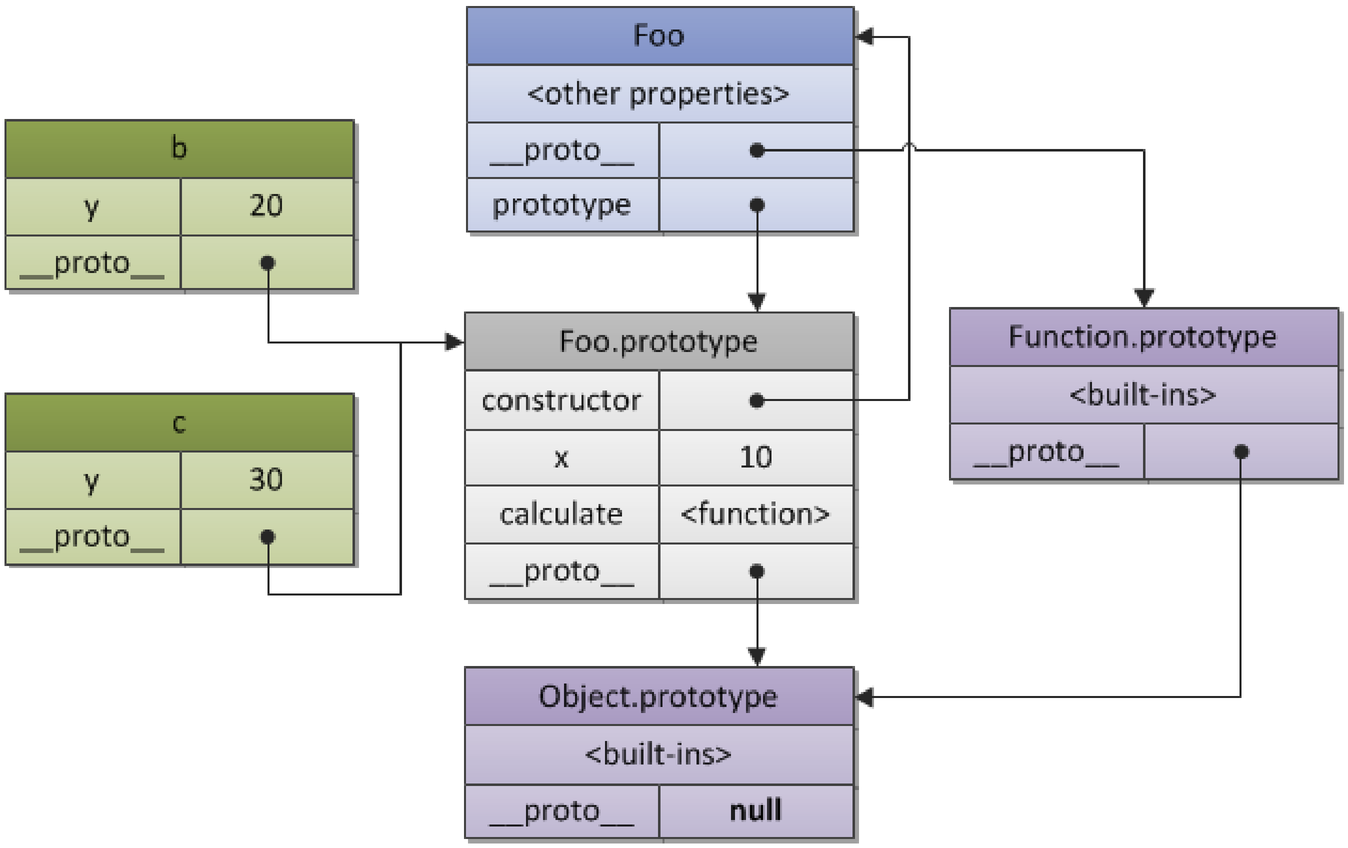


Figure 3. A constructor and objects relationship.

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:

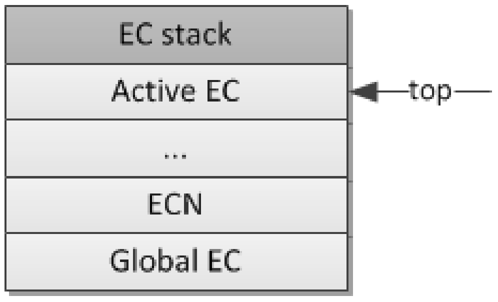


Figure 4. An execution context stack.

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:

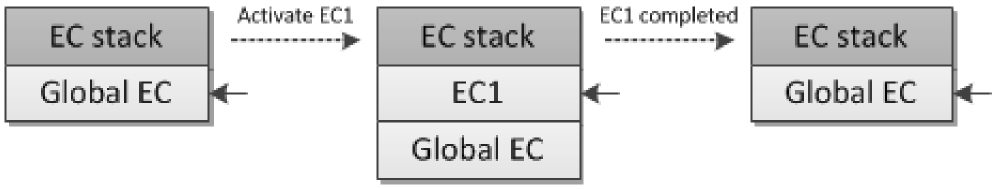


Figure 5. An execution context stack changes.

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:

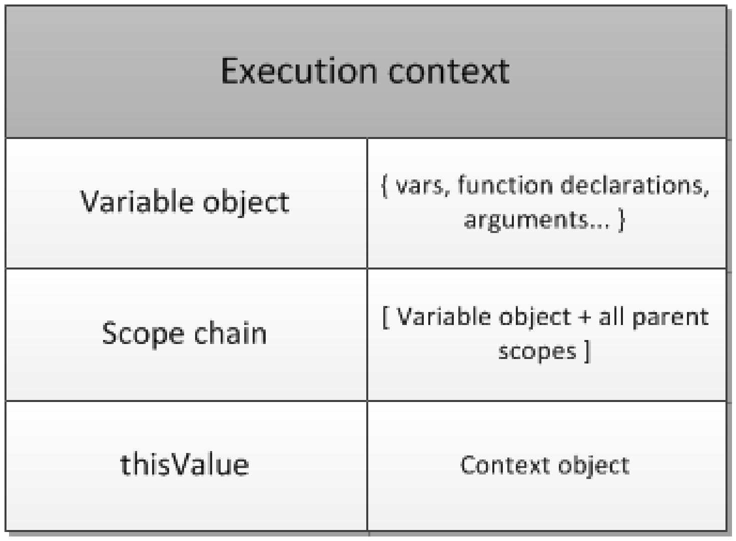

Execution context
Variable object
{ vars, function declarations, arguments... }
Scope chain
[Variable object + all parent scopes]
thisValue
Context object

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:

```
var foo = 10;

function bar() {} // function declaration, FD
(function baz() {}); // function expression, FE

console.log(
  this.foo == foo, // true
  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:

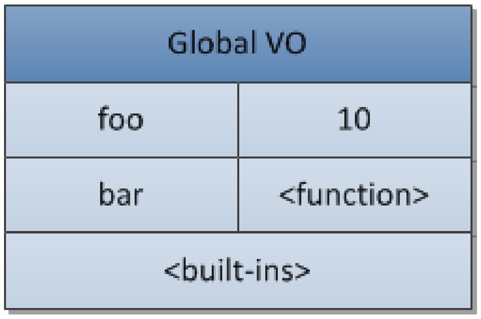


Figure 7. The global variable object.

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

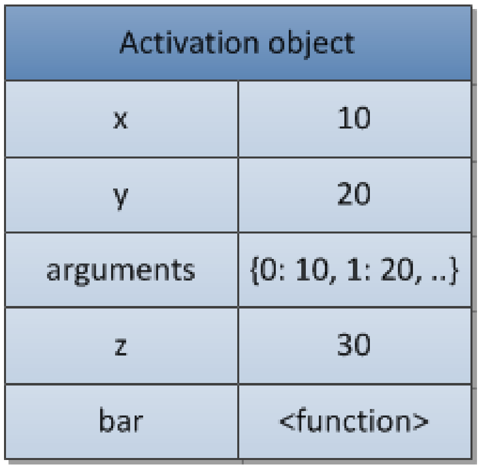
When a function is *activated* (called) by the caller, a special object, called an *activation object* is created. It's filled with *formal parameters* and the special `arguments` object (which is a map of formal parameters but with index-properties). The *activation object* then is used as a *variable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

```
var x = 10;

(function foo() {
  var y = 20;
  (function bar() {
    var z = 30;
    // "x" and "y" are "free variables"
    // and are found in the next (after
    // bar's activation object) object
    // of the bar's scope chain
    console.log(x + y + z);
  })();
})();
```

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):

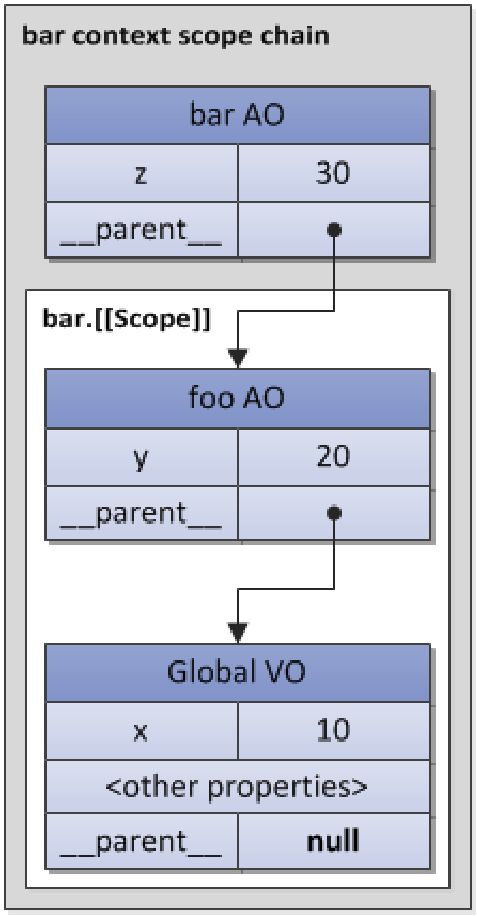


Figure 9. A scope chain.

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;

// in SpiderMonkey global object
// i.e. variable object of the global
// context inherits from "Object.prototype",
// so we may refer "not defined global
// variable x", which is found in
// the prototype chain

console.log(x); // 10

(function foo() {

  // "foo" local variables
  var w = 40;
  var x = 100;

  // "x" is found in the
  // "Object.prototype", because
  // {z: 50} inherits from it

  with ({z: 50}) {
    console.log(w, x, y , z); // 40, 10, 30, 50
  }

  // after "with" object is removed
  // from the scope chain, "x" is
  // again found in the AO of "foo" context;
  // variable "w" is also local
  console.log(x, w); // 100, 40

  // and that's how we may refer
  // shadowed global "w" variable in
  // the browser host environment
  console.log(window.w); // 20

})();
```

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

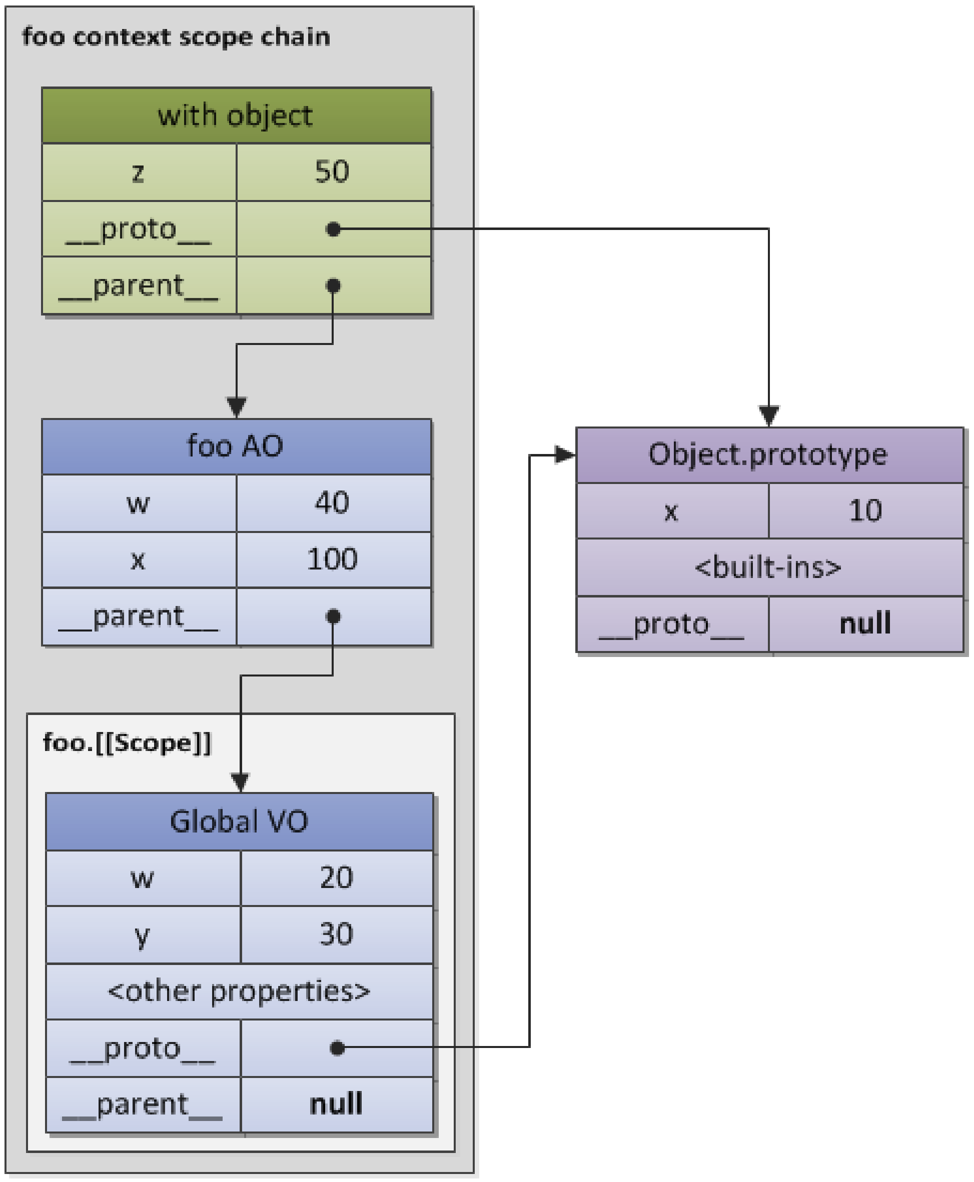


Figure 10. A "with-augmented" scope chain.

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:

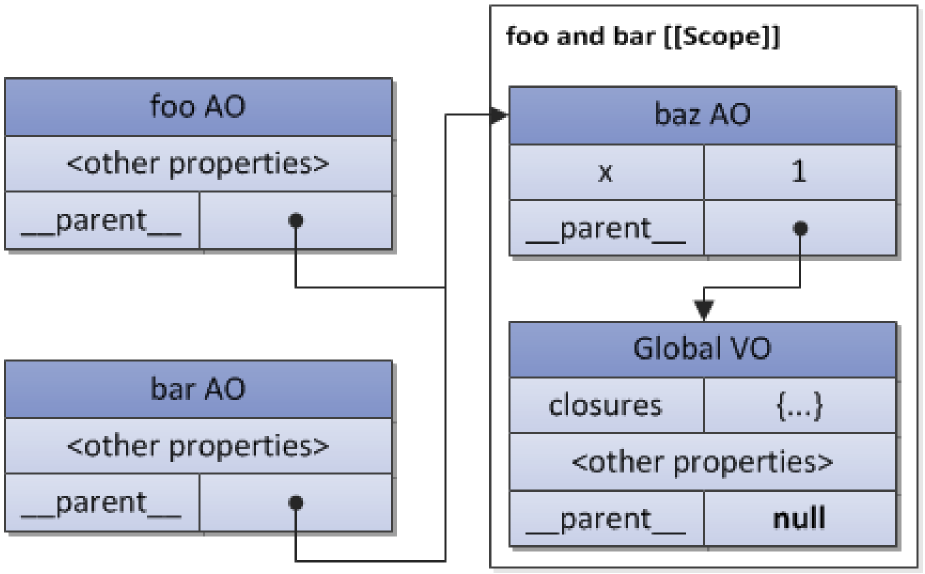


Figure 11. A shared [[Scope]].

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*):

```
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

Tweet Like 227 G+1 130 Share 104 Share

Tags: [[Scope]], Activation object, ECMA-262-3, ECMAScript, execution context, JavaScript, OOP, Scope chain, this, Variable object

This entry was posted on September 02nd, 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.

« Note 2. ECMAScript. Equality operators.
ECMA-262-5 in detail. Chapter 3.1. Lexical environments: Common Theory. »

117 Comments:

1. justin #permalink
13. September 2010 at 21:03

   Too late now
   I will carefully read it again tomorrow
   Thanks for excellent article

2. Josip Maras #permalink
13. September 2010 at 22:42

   Great work!

   Thanks

3. tim #permalink
14. September 2010 at 02:32

This is such a great series!

4. Simon Charette [#permalink](#)
   14. September 2010 at 05:29

   As usual you managed to make ECMA less obscure to me.

   Thanks!

5. leoner [#permalink](#)
   14. September 2010 at 09:00

   Great !
   Thanks

6. Benedikt [#permalink](#)
   14. September 2010 at 12:50

   Большое спасибо

   Wirklich sehr hilfreich, da sehr verständlich!

7. neurostep [#permalink](#)
   14. September 2010 at 16:27

```
(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
);
```

   Простите, может я что-то не допонимаю, но baz не будет видна, и консоль выдаст ошибку, нет?

8. AngusC #permalink
14. September 2010 at 19:41

Dmitry – an excellent tour of many of the language fundamentals. This will be very useful to any developer looking to move into advanced JavaScript. I particularly liked your explanation of static vs dynamic scoping.

One small thing: In "Execution context stack" you say (a couple of times) that uncaught exceptions "exist" the stack. I think you meant to say "exit"

9. 独孤逸辰 #permalink
14. September 2010 at 20:12

excellent !! very clear !!

10. Dmitry A. Soshnikov #permalink
14. September 2010 at 22:01

Thanks all, glad to see more people interested in deep JavaScript.

@**Benedikt**, Froh, dass war hilfreich 😉

@**neurostep**, да, была опечатка, спасибо. Должна быть простая функция, без автозапуска.

@**AngusC**, yeah, thanks Angus; fixed those typos.

Dmitry.

11. Allen #permalink
17. September 2010 at 18:33

Hi Dmitry!

Thank you for this just great article. And many thanks lots for the awesome javascript blog!

12. Сергей #permalink
18. September 2010 at 14:55

Отличная статья! Спасибо за Ваш труд!

Ваш ресурс по JavaScript — лучший. Очень жаль, что Вы больше не пишите на русском.

13. gniavaj [#permalink](#)
18. September 2010 at 17:17

hi~ Dmitry

do you have any article about the new operator?

I don’t quite understand the new operator’detail.

what will the js-engine do when i use the new operator create an object?

and what will happen when there is a return in the constructor(i.e a function)

14. [Dmitry A. Soshnikov](#) [#permalink](#)
18. September 2010 at 17:57

@**Сергей**,

спасибо. Да, возможно будет перевод этой статьи. Кстати, Вы тоже можете заняться. Из этой статьи тогда будет ссылка на внешний перевод.

@**gniavaj**,

> do you have any article about the new operator?

Yes, `new` in detail is explained in the [Chapter 7.2. OOP. ECMAScript implementation](#). For all related with OOP features I recommend reading the whole chapter.

> what will the js-engine do when i use the new operator create an object?

As [written in this lecture](#), a function applied to `new` operator, produces objects by specified pattern. The “pattern” here means that the function may assign to `this` instance properties (i.e. all objects created via the constructor have the same set of properties created via assigning to `this` in the constructor). Also, the constructor sets the prototype of newly created objects.

The complete algorithm of objects creation may be found again in the chapter 7.2 — in the [Constructor](#) section (and in particular in its subsection — [Algorithm of objects creation](#)).

In addition you may be interested in the algorithm of *functions* (i.e constructors themselves) creation; it may be found in the chapter 5 — [Algorithm of function creation](#).

> and what will happen when there is a return in the constructor(i.e a function)

By default (if there is no explicit `return` in a function) the *simple* function returns `undefined`.

However, if the function is applied as a constructor (i.e. with `new` operator), then it by default returns a *newly created object*. The same may be done via explicit return of `this`, i.e. the newly created object.

If the constructor returns *another object* — then this *object* is the result of the construction (and `this` value, i.e. the original created object is ignored).

If the function returns some *primitive value*, then this primitive is ignored and `this` value instead used.

This is briefly. In detail you may again found all this in the chapter 7.2 and mentioned above algorithm of objects creation.

Dmitry.

15. gniavaj #permalink
18. September 2010 at 19:15

Thanks!!!

16. lenel #permalink
25. September 2010 at 07:29

So Great Job~ Thanks.

17. lenel #permalink
25. September 2010 at 13:34

Dear Soshnikov. I have a question.
Is it possible to build a safe sandbox use "with" statment,that prevent some JS fragment access the DOM or BOM directly without change the original content of js?

Like :

```
with({window:winWrapped,document:docWrapperd,...}){

}
```

It may helps a lot for the safety of 3rd party content.
Hoping for your sugesstion~

18. Dmitry A. Soshnikov #permalink
25. September 2010 at 18:05

@lenel

> Is it possible to build a safe sandbox use “with” statment,that prevent some JS fragment access the DOM or BOM directly without change the original content of js?

`with` statement just adds its object to the front of the active context’s scope chain. If in your case `winWrapped` and `docWrapperd` would be the special *proxies* which trace some real objects (in this case `window` and `document`), then you may use `with` just to make manipulating of properties syntactically the same as you’d use real `window` or `document`.

However, currently we *don’t have* real *catch-all proxies* — they should appear only in ES6.

So *at the moment*, even if you’ll be able to use in the block of `with` statement something like `document.domain = '...'` (where `document` is your `docWrapperd`) it will do nothing — you’ll just change the property of your `docWrapperd` object, but not trace the real `document` object.

Additionally, *var*s and *FD*s defined within the `with` statement, *are not local for the with block*. Also with statement is obsolete and is removed from the ES5 *strict mode*. So possibly you’ll want to make a sandbox using a function context:

```
(function (window, document) {
  var auxiliaryData = 10;
  ...
})(winWrapped, docWrapperd);
```

But repeat, if you want exactly to prevent some direct access, e.g. *to trace the access*, you needed some *catch-all mechanism* which currently isn’t available neither in JS, nor in DOM layers.

P.S.: by the way, catch-all proxies are already available in nightly builds of Firefox 4.*beta. So you may install it and try to implement your wrappers with tracing.

The simpliest implementation:

```
var docWrapped = Proxy.create({
  get: function (rcvr, name) {
    console.log('Get:' + name);
    return document[name];
  },
  set: function (rcvr, name, val) {
    console.log('Set:' + name);
    return document[name] = val;
  }
}, Object.getPrototypeOf(document));

with ({document: docWrapped}) {
  document.foo = 10; // Set: foo
  console.log(document.foo); // Get: foo 10
}
```

However, always remember, that *host objects* (such as `document`) may not obey laws of *native objects* and *native methods* (such as *Object.getPrototypeOf*, etc.), so be careful with working with host objects.

Dmitry.

19. lenel [#permalink]
26. September 2010 at 11:49

Большое спасибо
Thanks very much~ ,Hope this series become a book in future.
More and more ES programmer will get benefit from it.
Not like other popular languages, ES lack base theory evangelist so.

20. galileo_galilei [#permalink]
7. October 2010 at 00:48

Well done and thank you for your effort Dmitry! U've put some very good tutorials in here.
Greeting from Romania

21. n-miyo [#permalink]
7. October 2010 at 02:45

Hi Dmitry,

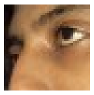
Thank you for great article! I love it. In figure 9, I wonder 'y' in box 'foo AO' should be 20 instead of 30. Thanks.

22. Dmitry A. Soshnikov [#permalink]
7. October 2010 at 13:19

@**galileo_galilei**, thanks, and send my greetings to Romania 😉

@**n-miyo**, hm, what a typo! Yeah, thanks, I'll fix it. Be careful also for 2 and 3 figures — there property "y" should have value 30, not 20. It will be fixed too soon.

Dmitry.

23.  Sumon [#permalink]
8. October 2010 at 07:54

It's really great and awesome post. Thanks for share with us !

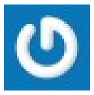
24. Qrilka [#permalink]
27. October 2010 at 00:40

Большое спасибо за статью, Дмитрий.
В предложении “The recent such a mistake was e.g. in this book (thought, the mentioned chapter of the book is quite good).” мне кажется вместо thought должно быть though.

25. Dmitry A. Soshnikov [#permalink]
27. October 2010 at 10:57

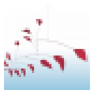
@**Sumon**, yep, thanks.

@**Qrilka**, да, опечатка, спасибо; исправил.

26.  Sudheer [#permalink]
14. November 2010 at 01:00

This is the best and succinct presentation of advanced JavaScript concepts that I came across. This explained many intricacies of JS and I could relate these concepts to similar in Lua. Execution Context was tricky initially but imagining it as a C Call frame turned into a dynamic object (simplification of course) helped.

Highly appreciate your effort and thanks for sharing.

27.  BelleveInvis [#permalink]
24. November 2010 at 16:13

Am I allowed to have a better Chinese translation? The given translation is not good. The result will be posted on [[typeof.net]].

28. Dmitry A. Soshnikov [#permalink]
24. November 2010 at 17:18

@**BelleveInvis**

Yes, I think it’d be good to have an alternative translation version of the same language (we already have for some chapters of the ES3 series).

Inform me when you’ll be ready with your version, I’ll give a link.

Dmitry.

29. Поросенок Петр #permalink
5. January 2011 at 18:36

Чувак – ты просто супер крут, спасибо!

30. eengel #permalink
17. January 2011 at 06:07

Hi Dmitry, great works!
Here I have a question: In figure 3, we see Foo have two properties: __proto__ and prototype. I tested them and like you mentioned, they're different. But I'm confused why we must have both of them as kinds of "prototype"?
Thanks!

31. [Dmitry A. Soshnikov](#) #permalink
17. January 2011 at 19:06

@**eengel**

> Here I have a question: In figure 3, we see Foo have two properties: __proto__ and prototype. I tested them and like you mentioned, they're different. But I'm confused why we must have both of them as kinds of "prototype"?

The real prototype of an object as said is the `__proto__` property. So regarding figure 3, prototype of `Foo` is `Function.prototype`.

However, as also mentioned, a constructor does two things: (1) creates objects and (2) sets the prototype of the newly created objects.

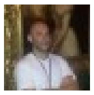
And exactly this prototype object which is set as a `__proto__` of the newly created objects is stored in the `prototype` property of the constructor.

Notice (on the same figure 3), `Foo` is a constructor, `Function` is also a constructor, `Object` is a constructor — and they all have property `prototype`.

`Foo` constructor created objects `a` and `b` and set their `__proto__` to `Foo.prototype.`

`So from all this, again: __proto__` is a prototype of an (any) object (in the specification it's called `[[Prototype]]` as said). And `ConstructorFunction.prototype` is a prototype of those object which are created by the `ConstructorFunction`.

Dmitry.

32.  Igor Ganapolsky [#permalink](#)
18. January 2011 at 02:56

Definitely a good resource on JS execution context. I haven’t seen a better explanation of it anywhere else.

33. [eminemence](#) [#permalink](#)
4. July 2011 at 11:26

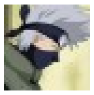
I tried these concepts in FF & Firebug, but somehow it gives me an error in Object.prototype. What tools do you use to check these concepts, as in which browser and debugger?

34. [eminemence](#) [#permalink](#)
5. July 2011 at 11:20

Answer to my above question is :
In JS we need to do something like this
b.prototype = a;
and the call to the calculate function would be like this:
b.prototype.calculate();

35. Kiran [#permalink](#)
8. July 2011 at 18:48

Hats off to the tutorial.. it clearly reset my fundamental thinking of JS programming

36.  el garch [#permalink](#)
4. August 2011 at 19:41

Great Job, thanks for the nice tutorial

37. Ding [#permalink](#)
12. August 2011 at 00:28

thank you Dmitry! another gem

38. Jerry [#permalink]
11. September 2011 at 14:18

Dmitry, very nice article – thanks!

I have a question regarding how the scope chain is handled when *inner functions* reassign *outer* functions.

The following example illustrates my question:

```
var outer = function(){
// lets call this scope [[Scope1]]
// note that NO local identifiers added to [[Scope1]]

    if( /* do some test */){

        // new function reassigns outer function
        outer = function(){
            // lets call this scope [[Scope2]]
        };

    }
    else{

        // other logic that does NOT introduce local identifiers

    }

};
```

I understand that if *outer* had defined local identifiers (variables or function declarations), then when the inner function reassigns *outer*,
the scope chain of it would include [[Scope1]]. So after reassignment, the Scope Chain when *outer* is called would be:

```
Scope Chain of outer after reassignment = AO + [[Scope2]] + [[Scope
```

But in the specific case above, *outer* did not define local identifiers and [[Scope1]] is essentially "empty" since built-in identifiers like *arguments* and *this* would be shadowed by the same named identifiers of the *inner function*.
In such case, is [[Scope1]] omitted from the Scope chain after reassignment since it does not contain *reachable* identifiers?

What I am trying to accomplish is a memory performant method of reassigning functions at execution time. The assignment, is of course, dependent on conditions not available at runtime.
If [[Scope1]] persists, I will probably resort to the following more runtime memory consuming method.

```
var outer = function(){
```

```
// note that NO identifiers added to the local scope

    if( /* do some test */){
    outer = outer.test1; // new function reassigns outer function
    }
    else{
    outer = outer.test2; // new function reassigns outer function
    }

};

outer.test1 = function(){};
outer.test2 = function(){};
```

The overall goal is to have *outer*'s scope chain (after being reassigned) to be: AO + [[Scope2]].

#permalink

39. Dmitry A. Soshnikov
11. September 2011 at 18:52

Jerry

> In such case, is [[Scope1]] omitted from the Scope chain after reassignment since it does not contain reachable identifiers?

By the spec — no; i.e. theoretically a function just captures parent scope chain as its `[[Scope]]` property regardless some conditions. However, in practice, yes, many implementations provide sensible optimizations capturing only used free variables (i.e. if none of parent variables is used, such functions even are not closures).

However, `eval` can break such optimizations, since in this case it's not known in advance which bindings will be used inside `eval`. I described this case in detail here and this part of lexical environment articles (notice in the later link that Python e.g. in contrast with JS doesn't worry about `eval` at all in this case).

#permalink

40. eric
25. September 2011 at 19:40

Guy, you already did great job.Nice articles.
I was read finished all related article, really need say "Thanks a lot", you've been resolved lots of problems confused me.
And i will continue to read those again, i know i still has lots misunderstanding.

#permalink

41. Crystal
3. November 2011 at 18:05

thanks from the deep bottom of my heart~~~thanks for ur works~~

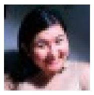
42. kai [#permalink]
23. November 2011 at 16:34

Great Job of explaining a really confusing topic.

The bad thing is, that I now may have to review a lot of my code for optimisations.

thx again

Kai

43.  Hannah [#permalink]
26. December 2011 at 18:41

Thank you so much for posting this! This is exactly what I needed in JavaScript. 🙂 You are a hero!

44. Chris [#permalink]
6. March 2012 at 14:17

Excellent work!

Thank you so much.

45. Sorin Jucovschi [#permalink]
19. March 2012 at 08:32

Thank you for the great article. I finally understand closures and scope.

46. gboyraz [#permalink]
20. April 2012 at 01:11

Excellent javascript articles! Thanks a lot

47. Cesar [#permalink]

27. April 2012 at 04:11

Excellent explanation. Thanks!

#permalink

48. kevin wang
25. May 2012 at 00:03

Hi, Dmitry,
From above, the __proto__ property is equal to the prototype property of Function object. what is the __proto__ property of Function object for?

#permalink

49. João Ferreira
1. July 2012 at 19:04

Thank you for the great article about ECMAScript.

#permalink

50. Alexei Ledenev
11. July 2012 at 03:47

Dmitry,
Thank you for great article. This is what I was looking for: just removes all confusion when starting to learn JS.

#permalink

51. Gaurav
10. October 2012 at 23:54

Thanks Dmitry. This is so far the best resource I ever found for learning JavaScript.

#permalink

52. kingysu
17. October 2012 at 19:17

I want to translate your articles into Chinese. Could you allow me to do so? Thank you.

#permalink

53. Dmitry Soshnikov
18. October 2012 at 19:34

@kingysu

Yeah, sure, send me the link when it will be finished. Also you might want to check already existing several Chinese translations of these articles.

[#permalink](#)

54. piglite
13. November 2012 at 22:38

Hi Dmitry:
Thanks a lot for you answered my last question several days ago!
And, There is a question about prototype.
if I hava a object `a` with a property, which name is "prototype", like below:

```
var a={prototype:{b:20}};
```

when I use `a.prototype` in the code block, which one should be taken? the prototype object, `a.prototype` or the value of property, `{b:20}`?
and step further:

```
var a={prototype:{}};
a.prototype=a.prototype;
```

At the last sentence, I set the prototype object to the value of object `a`'s property, or I set the prototype object to empty object?

[#permalink](#)

55. [Dmitry Soshnikov](#)
14. November 2012 at 21:42

@piglite

The explicit `prototype` property makes sense *only* for *functions* (in this case this property refers to the prototype of the *objects* which are *created* by this function).

And for any other object, the explicit `prototype` property is *just the simple property*, it's *not* the object's prototype. The object's prototype is, as is said in this article, the implicit property `__proto__`:

```
var F = function() {};

var f = new F;

console.log(f.__proto__ === F.prototype); // true
```

More detailed explanations are in [chapter 7.2 ES3](#) and also in this article above.

[#permalink](#)

56. piglite
14. November 2012 at 22:11

@Dmitry Soshnikov

thanks you reply, but I still confused that how does the implementation to tell the property of object “a” or a.prototype object, when it face the same writing way “a.prototype”?

and please read the code as below:

```
var Model = {
  prototype: {
    init: function() {}
  },
  create: function() {
    var object = Object.create(this);
    object.parent = this;
    object.prototype = object.fn = Object.create(this.prototype);
    return object;
  }
};
```

when use the `create` method to creat a new object:

```
var A = Model.create();
```

and which should be the `A.prototye` object? the property of Model object, mean the object of `{init:function(){}}` or the Model’s really prototype object, mean `Model.prototype`?

57. piglite
14. November 2012 at 23:43

#permalink

@Dmitry Soshnikov

Oh, I know what is your mean, finally!
When a function was used as constructor, the value of it’s prototype property should be referenced by the new `object._proto_`, in other cases, the prototype is just a property.
So, the sample of mine,

```
object.prototype = object.fn = Object.create(this.prototype);
```

mean the value of `object.prototype` property is a anonymous object which was created by Object’s create method, and the `object.prototype.__proto__` is the value of Model’s prototype property, `{init:function(){}}`,depend on the create method ways.
Is it right?

58. Dmitry Soshnikov
14. November 2012 at 23:54

#permalink

@piglite

Yes, that's correct.

59. Steven Guan
24. December 2012 at 23:10

#permalink

Hi Dmitry,
I tried following code. It seems function not always save the parent scope(not sure if I misunderstand the term :)). If a context is live in context stack, function just keep reference other than copy.

```
function foo4(p) {
  var x4 = p;
  function f1(p){
    var y4 = p;
    return function b1(){
        console.log("in b1 x4: " + x4);
        console.log("in b1 y4: " + y4);
    };
  };

  var r_f1 = f1(25);
  x4= x4*100;
  var r_f2 = f1(35);

  r_f1();
  r_f2();

  return f1(7);

};
console.log("foo4");
var r_foo4_1 = foo4(10); // x4 will be 1000 other than 10 and stor
var r_foo4_2 = foo4(20); // x4 will be 2000 other than 20 and stor

console.log("call r_f4_1()");
r_foo4_1();           //x4 is 1000
console.log("call r_f4_2()");
r_foo4_2();           //x4 is 2000
```

Am I right or miss something? Thanks.

60. Dmitry Soshnikov
26. December 2012 at 00:32

#permalink

@Steven Guan

Yes, that's correct. A function never copies the parent scope, it's always the reference is kept to it (regardless whether it's still active or finished). As long as there's at least one reference to the scope object it's kept in memory (this is how closures work — they just keep this reference and the object is still alive even after the context ends).

61. Steven Guan #permalink
26. December 2012 at 23:10

@Dmitry Soshnikov
If always reference is kept, the line 26, `r_foo4_1();`, should show `2000` than `1000`, am I right? In fact it show `1000`.

62. Dmitry Soshnikov #permalink
26. December 2012 at 23:26

@Steven Guan

No, *two separate calls* to `foo4`, create *two different* *activation objects* with their own `x4` inside. So the correct result is `1000` and `2000`.

63. Steven Guan #permalink
27. December 2012 at 23:50

Thanks 🙂

64. bhuvan #permalink
3. January 2013 at 07:19

if we are using someone else javascript library how i am suppose to see the eligible property of a particular object from that library.
i cant find any IDE that can do it.
OR do we go to lib code and see the eligible properties come back and use them.
Do people in the outside world do work this way ? (i have seen blog of some popular develop they seem to use vim,vi,emac)
or i am think all wrong here ?

i came from java,c++ background

65. bhuvan #permalink
19. January 2013 at 23:40

```
var Core= {};

(function init(){
    Core.foo = function(){
        console.log("in foo");
    }
})();
```

Core. "{now if i press some key here ..is there any IDE which can show me "foo" in suggestion list}"

66. Dmitry Soshnikov [#permalink](#)
20. January 2013 at 22:06

@**bhuvan** yes, there can many such IDEs (I cannot recommend one though, since use vim at the moment). But even simple Firebug console can do this. Or try running `Object.getOwnPropertyKeys(Core)` to introspect your object.

67. bhuvan [#permalink](#)
24. January 2013 at 23:50

```
var CORE = ( function init(my){
            //do stuff...
           })(CORE || {});
```

I see a lot of code do this "(CORE || {})".
What does it do ?

68. hdl253 [#permalink](#)
30. January 2013 at 06:55

```
for (var k=0; k < 3; k++){
    data[k]=function(){
        alert(k);
    };
}
for (var k=0; k < 3; k++){
    data[k](); // 0,1,2
}
```

how to understand the second loop counter "k".

69. Nikhil [#permalink](#)

13. February 2013 at 15:07

Hi Dmitry,
Thanks for the wonderful article. However I have a question about the constructors section and the figure 3 "A constructor and objects relationship".

I have a code snippets as follows

```
function A(y) {
    this.x = 25;
    this.y = y;
};

b = new A(20);
c = new A(30);
b.x = 10;
console.log(c.x)  // Prints out 25 as I expect.
```

Now if I change this code to be like your illustration to the below

```
function A(y) {
    this.y = y;
};

A.prototype.x = 25;

b = new A(20);
c = new A(30);

b.x = 10;
console.log(c.x);  // This also prints 25.
```

Given your illustration in Figure 3, `x` is now set on the prototype of `A`, to which the `__proto__` of both `b` and `c` are linked. So I would expect the `console.log(c.x)` to print out `10`.

Am I missing something here ?

Thanks again for the nice article

[#permalink](#)

70. [Dmitry Soshnikov](#)
13. February 2013 at 17:54

@**Nikhil**, [assignment operation](#) for simple properties always create *own property*. And this own property is said to *shadow* the property from the prototype (the property on the prototype still exist, you just don't reach it since it's found directly on the object).

So in both your case when you do `b.x = 10`, you create/modify *own* property on the `b` object. It doesn't modify `x` on the `A.prototype`.

```
function A() {}
A.prototype.x = 25;
```

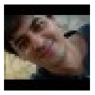
```

var b = new A;
var c = new A;

console.log(
  b.x, // 25
  c.x, // 25,
  A.prototype.x, // 25
  b.hasOwnProperty('x'), // false
  c.hasOwnProperty('x'), // false
  A.prototype.hasOwnProperty('x') // true
);

b.x = 10;

console.log(
  b.x, // 10
  c.x, // 25,
  A.prototype.x, // 25
  b.hasOwnProperty('x'), // true
  c.hasOwnProperty('x'), // false
  A.prototype.hasOwnProperty('x') // true
);

delete b.x;

console.log(
  b.x, // 25
  c.x, // 25,
  A.prototype.x, // 25
  b.hasOwnProperty('x'), // false
  c.hasOwnProperty('x'), // false
  A.prototype.hasOwnProperty('x') // true
);
```

71. nicholas

#permalink

17. February 2013 at 00:43

this is the best article about execute context in javascript I ever read, thanks!

72.  Jone

#permalink

20. February 2013 at 09:46

Just a question… what do you mean saying “cannot change value of this” ? this always refer to the actual execution context, but we can change the execution context by using the well know call/apply javascript functions. Am I right ?

73. David [#permalink](#)
20. March 2013 at 05:24

Thanks for this article, by the way I wanna ask you a question. Which tool did you use to draw diagrams in this article?

74. [Dmitry Soshnikov](#) [#permalink](#)
20. March 2013 at 11:21

@**nicholas**, thanks, glad it's useful.

@**Jone**

> `this` always refer to the actual execution context, but we can change the execution context by using the well know call/apply javascript functions

We should not confuse an [execution context](#) and the [this value](#) concepts.

The `this` can be called a *context object*, and an execution context is a different thing.

A function can be applied (called) using different context objects (as correctly you mention using e.g. `apply` or `call`). However, at the time of *running the execution context*, we cannot *assign* a new value to `this` object (i.e. we cannot do `this = {foo: 10};`, as we can do e.g. in Python with its `self` argument).

@**David**, it is MS Visio.

75. wangyinbin [#permalink](#)
10. April 2013 at 17:23

谢谢

76. fox road [#permalink](#)
21. April 2013 at 22:31

Hi Dmitry,
Thanks for the wonderful article. However I have a question about the constructors section and the figure 3 "A constructor and objects relationship".

I have a code snippets as follows

```
function Foo(y){
this.y=y;
this.say=function(){
alert(this.y+"tttttttttttttttt")}
}

Foo.prototype.says=function(){
alert(this.y+"i am the y property")}
function F(y){
this.y=y}


  F=Foo;// In this way i want the object test to use Foo's constru
//F.prototype=new Foo();  I don't know what the code here really m
//F.prototype=Foo.prototype; there is,when the say() method is inv
var test=new F(8);
test.say();
test.says();
alert(test.constructor+"constructor");
```

I wonder to know which way is the correct and why

77. Tom #permalink
19. May 2013 at 06:26

Thanks, It's real solve my many confusions when read js framework code. Used i think Javascript only can define variables, functions, change DOM elements.
谢谢，顶

78. Kevin Pauli #permalink
6. September 2013 at 11:28

Are these built-in shapes in Visio or something custom? I'd like to use something similar to document some JS design patterns.

79. Indrajeet Zala #permalink
24. September 2013 at 03:51

I must admit (I just can't leave without comments after reading for sure), this is an excellent post and undoubtedly the blog itself. Thank you Dmitry!

80. Dmitry Soshnikov #permalink
24. September 2013 at 22:26

@**Kevin Pauli**, yeah, it's MS Visio.

@**Indrajeet Zala**, thanks for reading, glad it's useful.

81. TanveerAli [#permalink](#)
26. November 2013 at 00:42

Awesome blog … Thank you

82. [JeremyWei](#) [#permalink](#)
14. December 2013 at 19:44

Hi, @Dmitry Soshnikov, here is the chinese edition: http://weizhifeng.net/javascript-the-core.html

83. [Dmitry Soshnikov](#) [#permalink](#)
17. December 2013 at 11:41

@**JeremyWei**, good job, will add the link.

84. hr [#permalink](#)
17. December 2013 at 20:38

Hi Dmitry,

I have a question about this example:

```
var data = [];

for (var k = 0; k &lt; 3; k++) {
  data[k] = function () {
    alert(k);
  };
}

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

I tested it gives 3 in all 3 cases.

However,earlier in the closure, you mentioned:

> 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**

If static scope is used, then in the code above function creation happens while the FOR loop is executed, the *free variable "k" will come from global VO*, which will have during the FOR loop values 0,1,2 for "k" wouldn't it- as the statement `data[k] = function(){ alert(k);}` is executed? So if it is static scope from that time , then when `data[0]`, `data[1]`…is executed we should be getting 0,1,2.
I feel either my understanding of what static scope is not correct or that AO of the anonymous function just connects it's implicit `__parent__` property Global VO BUT not take the value of K then(at the function creation), but it searches Global VO which is the scope at the function execution stage?

Look forward to hearing your explanation. Thanks in advance!

#permalink

85. Dmitry Soshnikov
18. December 2013 at 13:17

@**hr**

> or that AO of the anonymous function just connects it's implicit `__parent__` property Global VO BUT not take the value of K then(at the function creation), but it searches Global VO which is the scope at the function execution stage?

Yes, it's precisely this case. Notice, that it's still the static scope:

```
var k = 10;

var foo = function() {
  console.log(k);
};

(function() {
  var k = 20;
  foo(); // 10, but not 20
})();
```

I.e. function find the `k` in the scope chain of the context where it's created, but not where it's executed.

See this chapter for further explanations.

#permalink

86. hr
21. December 2013 at 10:59

Thx. Dmitry!

87. Pavels [#permalink](#)
17. January 2014 at 00:50

Thank you for the great article, it explained a lot.

88. unknown [#permalink](#)
16. March 2014 at 07:16

following example code on this article must be modified .

```
========================================================

var a = {
   x : 10,
   calculate : function(z){
      return this.x + this.y + z
   } // this.y isn't a property of "a or a's parents"
};



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

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

// call the inherited method
b.calculrate(30);  // NaN
c.calculrate(40);  // NaN


========================================================

var a = {
   x : 10
};

var b = {
   y : 20,
   __proto__ : a
   calculate : function(z){
      return this.x + this.y + z
   } // this.y is a property of parents "a"
};

var c = {
   y : 30,
   __proto__ : a
   calculate : function(z){
      return this.x + this.y + z
   } // this.y is a property of parents "a"
};

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

89. unknown
16. March 2014 at 07:24 [#permalink]

there is a mistake on comment of modified code by me .

```
// this.y is a property of parents “a”

=> // this.x is a property of parents "a"
```

90. Binh Thanh Nguyen
20. April 2014 at 23:11 [#permalink]

Thanks, nice post

91. Jackson
21. May 2014 at 18:14 [#permalink]

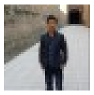
Hi Dmitry,
I’ve a question:
you said EC of eval will use the VO in the caller’s EC. What about the this value?

92. Dmitry Soshnikov
21. May 2014 at 21:51 [#permalink]

@**Jackson**, yes, the `this` value is reused from the calling context as well. Although notice, that in case of indirect eval, the `eval` runs in the global context.

93. Mukesh Kumar Saini
9. July 2014 at 03:54 #permalink

The best article on protoype i have seen,would like to read it again and again.

94.  Hong
24. July 2014 at 17:00 #permalink

@**Dmitry**, thanks for your great work.

95. Dmitry Soshnikov
24. July 2014 at 23:31 #permalink

@**Mukesh Kumar Saini**, @**Hong**, thanks for reading.

96. Andreas Bolm
30. July 2014 at 11:18 #permalink

Ok, really last try, here is a complete, corrected, corrected version of my previous postings — did I mention a preview button 🙂

I've a question about "downward funarg problem", and the following sentence:

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:

During translation I found the term: statically *saved at a function's creation*

The word *created* somewhat confuses me, because as we said, a function gets it's context at creation time, where context state, and scope chain among other things are created in *EC*. So the statically scope for `foo` refers ,in my humble optinion, to the scope of `foo` declaration, in this case the *global context scope.*

Therefore, in my German translation I tend to purpose the word function *declaration* instead of *creation*, this is maybe wrong, but as I've learned the `[[Scope]]` property is function related, too (not *EC* only, because *EC* will be destroyed at some point, and it's context state (*scope chain* property) get lost). Thus, some reference to `foo`'s definition context scope should(?) be stored at declaration time, too. Where else the function should know from which *scope* it steams from? I think the function can't know about it's *scope* at declaration without a reference at declaration time; but as I said earlier: *I'm maybe wrong.*

Another possibility for *static scope* mentioned, could be the current context of the *caller*, where `foo` as the argument to `funArg` is passed (and maybe evaluated (means created?)), in this case, the *static scope* would be the *global context scope* either.

Can you help me to clarify this, please?

[#permalink](#)

97. Gerald
1. August 2014 at 12:57

Hi Dmitry,

Thanks for sharing all of this extremely useful information. Thanks also for continuing to check back in to answer questions over the past 4 years!

Here's my question:

In figure 11, should Global VO include an entry for function *baz* (similar to figure 7, where Global VO included an entry for function *bar*)?

thanks

[#permalink](#)

98. [Dmitry Soshnikov](#)
1. August 2014 at 20:34

@Andreas Bolm

> The word *created* somewhat confuses me … I tend to purpose the word function *declaration*

In this case *created* I meant exactly *declared* (or *defined*).

```
var x = 10;

// Function "foo" is created
// (or defined, or declared).
function foo() {
  console.log(x);
}

// Function "bar" is created.
function bar(funArg) {
  var x = 20;
  funArg(); // 10 (from saved [[Scope]]), but not 20
}

// Function "bar" is called.
// Function "foo" is passed as an argument.
bar(foo);
```

In the example above at activation of the `funArg` (which is the `foo` function), the variable `x` is resolved in the lexically saved scope (i.e. 10), not in the scope of the caller (that would be 20).

99. Dmitry Soshnikov [#permalink]
1. August 2014 at 20:41

@**Gerald**, thanks, glad it's useful.

> In figure 11, should Global VO include an entry for function *baz*

Correct, it should Probably I should've put it explicitly there, but currently I think it's covered by the `<other properties>` section of that `Global VO` table on the figure 11.

100. Andreas Bolm [#permalink]
2. August 2014 at 16:21

@**Dmitry Soshnikov**

As you have shown in the above example, I've taken the necessary alterations in my translation accordingly.

I've to thank you for your time, and for sharing that excellent knowlege with us.

Must humble
Andreas Bolm

101. shadowhorst [#permalink]
9. October 2014 at 05:53

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

102. arch [#permalink]
19. February 2015 at 10:38

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

103. blajs [#permalink]
12. April 2015 at 01:05

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

104. Dmitry Soshnikov [#permalink]
12. April 2015 at 09:59

@**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.

105. Tom #permalink
3. June 2015 at 10:43

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.

106. Aflext Yang #permalink
8. August 2015 at 01:45

Great Article! Thanks a lot!

107. Anand Sonake #permalink
13. August 2015 at 21:37

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

108. manohar #permalink
19. August 2015 at 04:24

Fabulous

109. Sajid #permalink
5. October 2015 at 03:58

Thank You Dmitry,

Excellent series of articles.

It has clarified much of my misunderstanding.

110. [Dmitry Soshnikov](#) [#permalink](#)
5. October 2015 at 13:08

@**Sajid**, thanks, glad it's useful.

111. J Park [#permalink](#)
26. October 2015 at 04:46

Thank you for much for this article. super good.

112. Andrej Sagaidak [#permalink](#)
19. November 2015 at 01:27

Very clearly explained. Thank you so much Dmitry

113. plusman [#permalink](#)
6. December 2015 at 06:35

Excellent !

114. magicdawn [#permalink](#)
19. February 2016 at 18:29

Great work!

the ES6(aka ES2015) brings new problem.

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


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

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

115. Dmitry Soshnikov #permalink
20. February 2016 at 22:00

@**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);
}
```

116. magicdawn #permalink
21. February 2016 at 03:48

Thanks.

117. Faizal #permalink
2. March 2016 at 21:48

Awesome article.
Great!!

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

- Rafal Bartoszek on The quiz
  Nice quiz, #9 Like many people I like it the most :) #6 Just for my sense of completion...
- Dmitry Soshnikov on Тонкости ECMA-262-3. Часть 3. This.
  @Dmitriy не подскажите а почему алерт выдает undefined Потому что контекст, в котором создается объект не равен самому объекту. Только...
- Dmitriy on Тонкости ECMA-262-3. Часть 3. This.
  [js]var user = { firstName: "Василий", export: this }; alert(user.export.firstName); [/js] не подскажите а почему алерт выдает undefined,...

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