Dmitry Soshnikov in ECMAScript | 2010-03-07

ECMA-262-3 in detail. Chapter 3. This.

Read this article in: Russian, Chinese (version1, version2, version 3), Korean, French.

Introduction

In this article we will discuss one more detail directly related with execution contexts. The topic of discussion is the `this` keyword.

As the practice shows, this topic is difficult enough and often causes issues in determination of `this` value in different execution contexts.

Many programmers are used to thinking that the `this` keyword in programming languages is closely related to the object-oriented programming, exactly referring the newly created object by the constructor. In ECMAScript this concept is also

implemented, however, as we will see, here it is not limited only to definition of created object.

Let's see in detail what exactly `this` value is in ECMAScript.

Definitions

> `this` is a property of the . It's a special object in which context a code is executed.

```
activeExecutionContext = {
  VO: {...},
  this: thisValue
};
```

where VO is variable object which we discussed in the previous chapter.

`this` is directly related to the type of executable code of the context. The value is determined and is while the code is running in the context.

Let's consider these cases more in detail.

This value in the global code

Here everything is simple enough. In the global code, `this` value is the itself. Thus, it is possible to reference it indirectly:

```
// explicit property definition of
// the global object
this.a = 10; // global.a = 10
console.log(a); // 10

// implicit definition via assigning
// to unqualified identifier
b = 20;
console.log(this.b); // 20

// also implicit via variable declaration
// because variable object of the global context
// is the global object itself
var c = 30;
console.log(this.c); // 30
```

This value in the function code

Things are more interesting when `this` is used in function code. This case is the most difficult and causes many issues.

The first (and, probably, the main) feature of `this` value in this type of code is that here it is to a function.

As it has been mentioned above, `this` value is determined on entering the context, and in case with a function code the value can be .

However, at runtime of the code `this` value is , i.e. it is not possible to assign a new value to it since (in contrast, say, with programming language and its explicitly defined `self` object which can repeatedly be changed at runtime):

```
1  var foo = {x: 10};
2
3  var bar = {
4    x: 20,
5    test: function () {
6
7      console.log(this === bar); // true
8      console.log(this.x); // 20
9
10     this = foo; // error, can't change this value
11
12     console.log(this.x); // if there wasn't an error, then would be 10, not
13
14   }
15
16 };
17
18 // on entering the context this value is
19 // determined as "bar" object; why so - will
20 // be discussed below in detail
21
22 bar.test(); // true, 20
23
24 foo.test = bar.test;
25
26 // however here this value will now refer
27 // to "foo" - even though we're calling the same function
28
29 foo.test(); // false, 10
```

So what affects the variations of `this` value in function code? There are several factors.

First, in a usual function call, `this` is provided which activates the code of the context, i.e. . And the value of `this` is determined by the (in other words by the form how the function is called).

It is necessary to understand and remember this point in order to be able to determine `this` value in any context without any problems. Exactly the , i.e. the way of calling the function, influences `this` value of a called context .

(as we can see in some articles and even books on JavaScript which claim that `this` `this` `this` — what is description). Moving forward, we see that even normal global functions can be activated with which influence a different `this` value:

```
function foo() {
  console.log(this);
}

foo(); // global

console.log(foo === foo.prototype.constructor); // true

// but with another form of the call expression
// of the same function, this value is different

foo.prototype.constructor(); // foo.prototype
```

It is similarly possible to call the function defined as a method of some object, but `this` value will not be set to this object:

```
var foo = {
  bar: function () {
    console.log(this);
    console.log(this === foo);
  }
};

foo.bar(); // foo, true

var exampleFunc = foo.bar;

console.log(exampleFunc === foo.bar); // true

// again with another form of the call expression
// of the same function, we have different this value

exampleFunc(); // global, false
```

So how does the form of the call expression influences `this` value? In order to fully understand the determination of the `this` value, it's necessary to consider in detail one of the internal types — the `Reference` type.

Reference type

Using pseudo-code the value of `Reference` type can be represented as an object with two properties: (i.e. object to which a property belongs) and a in this base:

```
var valueOfReferenceType = {
  base: <base object>,
  propertyName: <property name>
};
```

Note: since ES5 a reference also contains property named `strict` — the flag whether a reference is resolved in the strict mode.

```
'use strict';

// Access foo.
foo;

// Reference for `foo`.
const fooReference = {
  base: global,
  propertyName: 'foo',
  strict: true,
};
```

Value of `Reference` type can be :

1. when we deal with an ;
2. or with a .

Identifiers are handled by the process of which is in detail considered in the Chapter 4. Scope chain. And here we just notice that at return from this algorithm there is a value of `Reference` type (it is important for `this` value).

Identifiers are variable names, function names, names of function arguments and names of unqualified properties of the global object. For example, for values on following identifiers:

```
var foo = 10;
function bar() {}
```

in intermediate results of operations, corresponding values of `Reference` type are the following:

```
var fooReference = {
```

```
  base: global,
  propertyName: 'foo'
};

var barReference = {
  base: global,
  propertyName: 'bar'
};
```

For getting the of an object from a value of `Reference` type there is `GetValue` method which in a pseudo-code can be described as follows:

```
function GetValue(value) {

  if (Type(value) != Reference) {
    return value;
  }

  var base = GetBase(value);

  if (base === null) {
    throw new ReferenceError;
  }

  return base.[[Get]](GetPropertyName(value));

}
```

where the internal `[[Get]]` method returns of object's property, including as well analysis of the inherited properties from a prototype chain:

```
GetValue(fooReference); // 10
GetValue(barReference); // function object "bar"
```

are also know; there are two variations: the (when the property name is correct identifier and is in advance known), or the :

```
foo.bar();
foo['bar']();
```

On return of intermediate calculation we also have the value of `Reference` type:

```
var fooBarReference = {
  base: foo,
  propertyName: 'bar'
};

GetValue(fooBarReference); // function object "bar"
```

So, how a value of `Reference` type is related with `this` value of a function context? — . The given moment is the main of this

article. The general rule of determination of `this` value in a function context sounds as follows:

> The value of `this` in a function context is provided (how the function call is written syntactically).
>
> If on the left hand side from the call parentheses `( ... )`, there is a value of `Reference` type then `this` value is set to the of this value of `Reference` type.
>
> (i.e. with value type which is distinct from the `Reference` type), `this` value is always set to `null`. But since there is no any sense in `null` for `this` value, it is converted to .

Let's show on examples:

```
function foo() {
  return this;
}

foo(); // global
```

We see that on the left hand side of call parentheses there is a `Reference` type value (because is an identifier):

```
var fooReference = {
  base: global,
  propertyName: 'foo'
};
```

Accordingly, `this` value is set to base object of this value of `Reference` type, i.e. to global object.

Similarly with the property accessor:

```
var foo = {
  bar: function () {
    return this;
  }
};

foo.bar(); // foo
```

Again we have the value of type `Reference` which base is `foo` object and which is used as `this` value at `bar` function activation:

```
var fooBarReference = {
  base: foo,
  propertyName: 'bar'
};
```

However, activating with , we have already other `this` value:

```
var test = foo.bar;
test(); // global
```

because `test`, being the identifier, produces other value of `Reference` type, which base (the global object) is used as `this` value:

```
var testReference = {
  base: global,
  propertyName: 'test'
};
```

> Note, in the strict mode of ES5 `this` value is not coerced to global object, but instead is set to `undefined`.

Now we can precisely tell, why the same function activated with , has also different `this` values — the answer is in different intermediate values of type `Reference`:

```
function foo() {
  console.log(this);
}

foo(); // global, because

var fooReference = {
  base: global,
  propertyName: 'foo'
};

console.log(foo === foo.prototype.constructor); // true

// another form of the call expression

foo.prototype.constructor(); // foo.prototype, because

var fooPrototypeConstructorReference = {
  base: foo.prototype,
  propertyName: 'constructor'
};
```

Another (classical) example of dynamic determination of `this` value by the form of a call expression:

```
function foo() {
  console.log(this.bar);
}

var x = {bar: 10};
var y = {bar: 20};

x.test = foo;
y.test = foo;

x.test(); // 10
y.test(); // 20
```

Function call and non-Reference type

So, as we have noted, in case when on the left hand side of call parentheses there is a value of `Reference` type but type, `this` value is automatically set to `null` and, as consequence, to the object.

Let's consider examples of such expressions:

```
(function () {
  console.log(this); // null => global
})();
```

In this case, we have object but not object of `Reference` type (it is not the identifier and not the property accessor), accordingly `this` value finally is set to global object.

More complex examples:

```
var foo = {
  bar: function () {
    console.log(this);
  }
};

foo.bar(); // Reference, OK => foo
(foo.bar)(); // Reference, OK => foo

(foo.bar = foo.bar)(); // global?
(false || foo.bar)(); // global?
(foo.bar, foo.bar)(); // global?
```

So, why having a which intermediate result should be a value of `Reference` type, in certain calls we get for `this` value not the base object (i.e. `foo`) but ?

The matter is that last three calls, , have already on the left hand side of call parentheses the value .

With the first case all is clear – there unequivocally `Reference` type and, as consequence, `this` value is the base object, i.e. `foo`.

In the second case there is a which , considered above, method of getting the real value of an object from value of `Reference` type, i.e. `GetValue` (see note of 11.1.6). Accordingly, at return from evaluation of the grouping operator — we still have a value of `Reference` type and that is why `this` value is again set to the base object, i.e. `foo`.

In the third case, , unlike the grouping operator, `GetValue` (see step 3 of 11.13.1). As a result at return there is already object (but not a value of `Reference` type) which means that `this` value set to `null` and, as consequence, to .

Similarly with the fourth and fifth cases — and call the `GetValue` method and accordingly we lose value of type `Reference` and get value of type ; and again `this` value is set to .

Reference type and null this value

There is a case when call expression determines on the left hand side of call parentheses the value of `Reference` type, however `this` value is set to `null` and, as consequence, to . It is related to the case when the base object of `Reference` type value is the activation object.

We can see this situation on an example with the inner function called from the parent. As we know from the second chapter, local variables, inner functions and formal parameters are stored in the of the given function:

```
1 function foo() {
2   function bar() {
3     console.log(this); // global
4   }
5   bar(); // the same as AO.bar()
6 }
```

The activation object always returns as `this` value — `null` (i.e. pseudo-code `AO.bar()` is equivalent to `null.bar()`). Here again we come back to the described

above case, and again, `this` value is set to .

The exception can be with a function call inside the block of the `with` statement in case if object contains a function name property. The `with` statement adds its object in front of scope chain i.e. the activation object. Accordingly, having values of type `Reference` (by the identifier or a property accessor) we have base object not as an activation object but object of a `with` statement. By the way, it relates not only to inner, but also to global functions because the `with` object higher object (global or an activation object) of the scope chain:

```
var x = 10;

with ({

  foo: function () {
    console.log(this.x);
  },
  x: 20

}) {

  foo(); // 20

}

// because

var  fooReference = {
  base: __withObject,
  propertyName: 'foo'
};
```

The similar situation should be with calling of the function which is the actual parameter of the `catch` clause: in this case the `catch` object is also added in of scope chain i.e. the activation or global object. However, the given behavior was recognized as a bug of ECMA-262-3 and is fixed in the new version of standard — ECMA-262-5. I.e. `this` value in the given activation should be set to global object, but not to `catch` object:

```
try {
  throw function () {
    console.log(this);
  };
} catch (e) {
  e(); // __catchObject - in ES3, global - fixed in ES5
}

// on idea

var eReference = {
  base: __catchObject,
  propertyName: 'e'
};

// but, as this is a bug
// then this value is forced to global
```

```
// null => global

var eReference = {
  base: global,
  propertyName: 'e'
};
```

The same situation with a recursive call of the named function expression (more detailed about functions see in Chapter 5. Functions). At the first call of function, base object is the parent activation object (or the global object), at the recursive call — base object should be special object storing the optional name of a function expression. However, in this case `this` value is also always set to :

```
(function foo(bar) {

  console.log(this);

  !bar && foo(1); // "should" be special object, but always (correct) global

})(); // global
```

This value in function called as the constructor

There is one more case related with `this` value in a function context — it is a call of function as the constructor:

```
function A() {
  console.log(this); // newly created object, below - "a" object
  this.x = 10;
}

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

In this case, the new operator calls the internal [[Construct]] method of the `A` function which, in turn, after object creation, calls the internal [[Call]] method, all the same function `A`, having provided as `this` value newly created object.

Manual setting of this value for a function call

There are two methods defined in the `Function.prototype` (therefore they are accessible to all functions), allowing to specify `this` value of a function call manually. These are `apply` and `call` methods.

Both of them accept as the first argument `this` value which is used in a called context. A difference between these methods is insignificant: for the `apply` the

second argument necessarily should be an array (or, the , for example, `arguments`), in turn, the `call` method can accept any arguments; obligatory arguments for both methods is only the first — `this` value.

Examples:

```
var b = 10;

function a(c) {
  console.log(this.b);
  console.log(c);
}

a(20); // this === global, this.b == 10, c == 20

a.call({b: 20}, 30); // this === {b: 20}, this.b == 20, c == 30
a.apply({b: 30}, [40]) // this === {b: 30}, this.b == 30, c == 40
```

Conclusion

In this article we have discussed features of the `this` keyword in ECMAScript (and they really are , in contrast, say, with C++ or Java). I hope article helped to understand more accurately how `this` keyword works in ECMAScript. As always, I am glad to answer your questions in comments.

Additional literature

10.1.7 – This;
11.1.1 – The this keyword;
11.2.2 – The new operator;
11.2.3 – Function calls.

Translated by: Dmitry A. Soshnikov with help of Stoyan Stefanov.
Published on: 2010-03-07

Originally written by: Dmitry A. Soshnikov [ru, read »]
With additions and corrections by: Zeroglif

Originally published on: 2009-06-28; **updated on:** 2010-03-07

Dmitry Soshnikov

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

Published

2010-03-07

Write a Comment

104 COMMENTS

← Older Comments

Newer Comments →

Hong

2014-07-23

@**Dmitry**, thanks for your detail explanation, reconsider your last comment again, I think I finally get it.

The first one without “var that = this;”, when calling “object.getNameFunc()()”, the left hand side of final call paranthesis (…) is ‘object.getNameFunc()’, which is one function object (that returned function), not a reference type, so ‘this’ value is resolved to ‘null’ -> ‘global’.

To the second one, when left hand side ‘object.getNameFunc()’ executed, ‘object.getNameFunc’ is one Reference type, so ‘this’ is resolved to its base ‘object’, and then assigned to ‘that’, and ‘that’ is finally statically captured by the returned closure.

I checked ECMA Spec, it seems Reference Type is one intermediate represent form of expression evaluation result, its introduction is purely for expository purposes.

In addition, I have another question, please help.

There is one note in section “11.1.6 The Grouping Operator”.

———————————————

NOTE
This algorithm does not apply GetValue to Result(1). The principal motivation for this is so that operators such as delete and typeof may be applied to parenthesised expressions.

Could you please give more explanation about this motivation? Is there any special to ‘delete’ and ‘typeof’? Thanks.

Regards.

Dmitry Soshnikov
2014-07-24

@Hong

> *which is one function object (that returned function), not a reference type, so ‘this’ value is resolved to ‘null’ -> ‘global’.*

Yes, absolutely correct.

> *Could you please give more explanation about this motivation? Is there any special to ‘delete’ and ‘typeof’?*

Yeah, if the grouping would apply `GetValue`, this wouldn’t work (usually `typeof` is used without parens, but just for the example):

```
1 | typeof(foo); // "undefined"
```

It shows correctly `"undefined"`, and with `GetValue` applied it would throw `ReferenceError` since would try to get the value on non-existing var.

Hong
2014-07-24

@Dmitry, thanks for your help:)

Hong
2014-11-23

@Dmitry

Please help, two more questions 🙂

In the section `Reference type and null this value`, for the last two points.

Q.1 for the first point

> *The similar situation should be with calling of the function which is the actual parameter of the catch clause: in this case the catch object is also added in front of scope chain i.e. before the activation or global object. However, the given behavior was recognized as a bug of ECMA-262-3 and is fixed in the new version of standard — ECMA-262-5.*

I still cannot understand why `this` is set to `_catchObject` was recognized as a bug? I am not clear why it is one bug, based on your previous explanation, I think it is reasonable to set `this` in catch clause to `_catchObject`. Why should `this` value be set to global object rather than `_catchObject`?

Q.2 for the second point

> *The same situation with a recursive call of the named function expression (more detailed about functions see in Chapter 5. Functions). At the first call of function, base object is the parent activation object (or the global object), at the recursive call — base object should be special object storing the optional name of a function expression. However, in this case this value is also always set to global:*

Based on your explanation, it is also reasonable to set `this` under such situation to `special object`, why `this` is always set to global?

Regards.

Dmitry Soshnikov
2014-11-24

@**Hong** those are special cases, where `this` value has to be set to global — to support idiom that calling a function is a simple form — `foo();` — should execute in with the

global `this`. So, it should be the catch object, and the object for the name in case of named function expression, but it was decided to pass the global.

xinwendashibaike
2014-12-15

Thanks a lot for this wonderful series of articles!

Hong
2014-12-16

@Dmitry

When we using following JS segments to achieve some effects, `this` will be resolved to `global window`.

```
...
  li.onmouseover = function () {
    setTimeout(function () {
      alert(this);
    }, 500);
  }
  ...
```

Whether following explanation is right? `the anonymous function in` **`setTimeout`** `is one inner function of` **`onmouseover`**`, when it exectutes, its left call object is` **`AO`** `of` **`onmouseover`** `so this is resolved to window.`

Thanks,
Hong

Dmitry Soshnikov
2014-12-19

@Hong

In this case we only pass the function to the `setTimeout`, we don't execute ourselves. It will be executed by the `setTimeout` later. And in general we don't actually know in it will be executed. It can be a simple call without explicit `this` (and `this` will be global), or potentially it could be some `.call(…)`. But in case of `setTimeout` it just executes the function w/o explicit context, so it is global.

Notice: `this` value is not related to how or where, or when a function is created. It relates only to the execution time: to the form how the function is called.

blajs
2015-02-24

Hi, Dmitry.
One question: when you say "**the form of call expression**" you mean how the function call looks visually, for example:

```
1 myArray[0]();
2 obj.foo();
3 foo();
4 obj.x.y();
```

These are all different forms of call expression, am I right ? Thanks

Dmitry Soshnikov
2015-02-25

@blajs

Yes, that is correct. Since even the simplest function can be called in many different forms:

```
1 function foo() {
2   console.log(this);
3 }
4
5 foo(); // first form
6 foo.prototype.constructor(); // second form, `this` is different
```

hari
2015-03-27

Hi Dmitry,

> *The activation object always returns as* `this` *value —* `null` *(i.e. pseudo-code* `AO.bar()` *is equivalent to* `null.bar()` *).*

Why does Activation Object return `null` ? Can you please explain?

Thanks,
Hari

Dmitry Soshnikov
2015-03-27

@hari,

This is how ES3 specifies it, and it's to support common pattern "if a function is called in this form — `foo();` , then `this` value should be global object" (unless it's called within a `with` statement which object has `foo` function). And if `this` is passed as `null` , it's automatically defaulted to global object.

In other words:

```
function test() {
  function foo() {
    console.log(this);
  }
  foo(); // global object
}
```

And this is because `foo()` -> `AO.foo()` -> `null.foo()` -> `foo.call(globalObject)` .

hari
2015-03-28

Thanks Dmitry! In general any inner function invocation , will have **this** as irrespective of the inner function is in a function or within a method in an object

hari

2015-04-03

Dmitry,
I would appreciate if you would look at these cases and tell me where my understanding has gone wrong.

Example 1
=========

```
1  var a = 20;
2  var x = function(){
3  this.y = 10;
4  };
5  var y ={
6       a:10,
7      get: function(){ return 'from y  a= ' + this.a}
8  };
9  x.prototype = y;
10 instX = new x();
11 console.log(instX.get());//10
```

In this case call form expression instX.get() is a reference type.
Here is my thought process:

Hence, according to the rule it is base object of instX- which I would think is global. if that is the case output would be this.a = global.a = 20. *But , that is not the output- it is 10.* It seems that a is resolved by looking at the __proto__ object of instX . instX object looks like this

```
1  instX
2        y:10
3        __proto__
4                a:10
5                get:
6                __proto__
```

I would appreciate, if you can explain , where i have gone wrong in my thought process.

Second example
==============

Also,

```
var a = 20;
var x = function(){
this.y = 10;
};
var y ={
     a:10,
    get: function(){ return 'from y  a= ' + a}
};
x.prototype = y;
instX = new x();
a= 30;
console.log(instX.get());//30
```

This is more of scope chain question.
in this case a seems to have been picked from global. The functionalContext of “get” method will be AO + global VO, is that the correct interpretation of the scope chain in the case of the method?
Thanks in advance!

Dmitry Soshnikov
2015-04-07

@hari

> *which I would think is global*

It’s a global , but not the itself. So in the call of `instX.get()` the `this` value is `instX` , and it has (from `y`) `a` as `10` . So your inheritance chain analysis is correct.

> *is that the correct interpretation of the scope chain*

Yes, your scope chain analysis is correct as well.

hari
2015-04-15

Dmitry,
Thanks !

hari
2015-04-15

Dmitry,
So in second example, why wouldn't inheritance chain lead to resolving `a` as `10` before scope chain? For identifier resolution, does scope chain have a higher priority than inheritance chain?

Dmitry Soshnikov
2015-04-17

@**hari**, activation objects do not inherit from anything (there were some implementations in the past where they did though), so `a` is resolved in the global.

One example when inheritance chain is considered is with using `with` statement: its object is added to the scope chain, and identifiers are resolved in it as well (including its prototype chain):

```
1  var a = 10;
2  Object.prototype.a = 20;
3
4
5  with ({x: 20}) {
6    console.log(a, x); // 20 (from Object.prototype), 20
7  }
8
9  delete Object.prototype.a;
10
11 with ({x: 20}) {
12   console.log(a, x); // 10 (from global), 20
13 }
```

David Lee
2016-01-01

Dmitry,

I have a question about below explanation in the article.

"this value is a property of the execution context:"

Shouldn't it be like "this is a property of the execution context:"? I think the property is not a value of 'this' but 'this'. 'thisValue' is the value of the property 'this'.

What do you think? If I misunderstood, please correct me.

Thanks.

David.

Dmitry Soshnikov
2016-01-02

@David Lee

> *Shouldn't it be like ?*

Yeah, good point. Thanks, I changed!

beginner
2016-01-29

Great great articles, thank you!

personally i think semicolon is better for understanding in " __catchObject – in ES3; global – fixed in ES5"

yezi
2016-02-16

hi,Dmitry
You mentioned in the above comments：

> *A caller — is a context which invokes (calls) some other context.*
> *In turn, a context which is being called is a callee.*

Is the above “a context” equivalent to functionContext or globalContext in ECStack?And the ‘arguments.callee’ also means that?if yes, is it means that the called function self is a context? It looks a little strange.

Dmitry Soshnikov
2016-02-16

@yezi,

> *Is the above “a context” equivalent to functionContext or globalContext in ECStack?And the ‘arguments.callee’ also means that?if yes, is it means that the called function self is a context?*

Well, in case of functions it’s easier to think about as functions themselves (in this case `arguments.callee` makes perfect sense, as well as legacy `arguments.caller`).

However, if you take e.g. global code, it also can call some function. And in this case — who is a caller? We don’t have any function that calls us, rather the function is called from the global . So you may normally say that a function is a caller, or its context is a caller — these can be used interchangeably.

yezi
2016-02-17

Dmitry,
Thanks!
I’m not good at English,so there may be some problems with my presentation.
I still have some doubts:
1)In case of functions, may I think about caller/callee as some context object(globalContext or functionContext) in ECStack?and does ECStack really exist in implementation,or it’s just a concept?
2)for your answer:

> *So you may normally say that a function is a caller, or its context is a caller — these can be used interchangeably.*

Does it mean that the function context is just a concept,and not a real object in implementation?otherwise,the function context should be a function instance when the function is called?

Dmitry Soshnikov
2016-02-20

@yezi

> *Does it mean that the function context is just a concept,and not a real object in implementation?*

No, the , and is something that is specified, so it exists in implementations.

So whenever we talk about JS code, and about of the JS code — we talk about executing a code of a . Take a look e.g. at the [[Call]] algorithm — there is a phrase, , or .

But (in user-code, not implementation, since user-code doesn't have access to the execution context), when calling a function, the function itself can be called as a (if it calls another function), or a (if it's being called).

In older versions there was even arguments.caller, but was removed, since non-standard.

yezi
2016-02-21

Dmitry,
Thanks for your answer!And I also hope that you can write a book on ECMA-262 in detail,because in China,there are few books to talk about the knowledge in your series of articles.

Dmitry Soshnikov
2016-02-21

@**yezi**, thanks, glad you found it useful!

Andrew
2016-07-26

Could you please explain why in the following example it doesn't see 'name' in FE scope and prints just an empty string?

```
var a = {
    foo: () =>; {
        console.log(name);
    }
};

(() =>; {
    var name = "2";
    a.foo();
})();
```

Dmitry Soshnikov
2016-08-05

@**Andrew**, the `var name = "2";` is defined in completely different scope, so `a.foo()` when is called don't see it (it's correct). The empty string is from the global `name` if you test in browser (it's `window.name`).

Jason
2016-11-02

Dimitry thank you for the article. I got confused when the reference type is not clear. For example.

```
function Prefixer(prefix) {
    this.prefix = prefix;
}
Prefixer.prototype.prefixArray = function (arr) { // (A)
    return arr.map(function (x) { // (B)
        // Doesn't work:
        return this.prefix + x; // (C)
    });
};
```

What is the value of this here. The implementation of map method is not clear.

Dmitry Soshnikov

2016-11-02

@Jason, the `map` method accepts also an optional second parameter, actually the `this` value.

```
return arr.map(function (x) { // (B)
  return this.prefix + x; // (C)
}, this); // <-- see, this is passed
```

Another alternative is to use the `bind` method of functions, which binds the `this` value:

```
return arr.map(function (x) { // (B)
  return this.prefix + x; // (C)
}.bind(this)); // bind the this
```

And one more (recommended these days) approach is just to use an arrow-function, which has a `this`, which is kinda "auto-bound":

```
return arr.map(x => { // (B)
  return this.prefix + x; // (C)
}); // this is lexical
```

If there is only one statement, can be written in the short way:

```
return arr.map(x => this.prefix + x); // this is lexical
```

The reason why `map` calls your callback in the global context, is because it's called as:

```
function map(array, callback, context) {
  let result = [];

```

```
  for (let k = 0; k < array.length; k++) {
    if (context) {
      // Call in passed context.
      result[k] = callback.call(context, array[k], k, array);
    } else {
      // Call in global context.
      result[k] = callback(array[k], k, array);
    }
  }

  return result;
}
```

Jason

2016-11-02

Dmitry, thank you very much. Is there anyway to look into how map method is implemented other than look at the ecma specification?

Dmitry Soshnikov

2016-11-02

@Jason, yes, you can take a look at any open-source implementation, e.g. V8 engine. Also, this polifyll from MDN reflects the spec.

dagolinuxoid

2017-01-19

Brilliantly! Absolutely massive!! | Most others authors explanations, that I’ve heard before are just pale in comparison to what I have known here.

Dmitry Soshnikov

2017-01-19

@dagolinuxoid, thanks, glad the material is useful!

Xue Shihan
2017-03-23

Hi, I'm not sure if you can receive my message, but I really need your help. I am learning JavaScript by myself now. When I find your blog, I feel so happy. Great articles! I have understand so many concepts, but still, there are also some questions I have no answers. I read your articles again and again. Finally, I decide to ask for your help directly.

Please see my question below.

You say, "in an usual function call, this is provided by the caller which activates the code of the context, i.e. the parent context which calls the function."

I don't understand what the "caller" means here. In another post, I see "A context which activates another context is called a caller." So a caller is a context? Am I right? And if "context" here equals "execution context"?

I will appreciate your help. Thank you in advance.

Dmitry Soshnikov
2017-03-23

@Xue Shihan,

> *So a caller is a context?*

Yes, depends on which level we discuss the : if we talk about , yes, "caller" is an (execution) context. At user-code level, a might be a . In older implementations `arguments.caller` was a reference to a function which calls a running function.

Xue Shihan

2017-03-23

@Dmitry Soshnikov,

Thank you so much! Your blog helped me a lot!

Vincent

2017-03-31

Hi Dmitry,

I have read all the 8 chapters, and I understand the two phases of execution context, and the real reasons of variable hoisting.

Inspired by leoner's question on 2010-05-07, I tried the following piece of code in Chrome (Version 56.0.2924.87) and Firefox(Version 51.0.1 (32-bit)):

```
with ({x: 50}) {
  function foo() {
      console.log(x);
  }
  foo(); // 50, i understand why it is 50, because {x: 50} is prepended t
}
var x = 20;

foo(); // why it is still 50 ?
```

As you said in your reply to leoner, the function is created on entering execution context in case of function declaration, and according to chapter 4, [[Scope]] is determined then. Therefore, in my understanding, the foo function is hoisted, and the code can be transformed:

```
var x; // undefined

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

with ({x: 50}) {
  foo();
}

x = 20; // modified as to be 20

foo();
```

According to all the principles I have learned from your articles, when being created, the foo's [[Scope]] only contains the global object, and after the with block exits, the with object is removed from the front of the scope chain, so there is only global object in the chain onward, and the x identifier should be resolved as the x variable on global object, which is 20 when the foo function is called outside the with block. So, I expected the last call would output 20, but I got 50.

Could you explain why it is 50? Thank you.

Vincent

Dmitry Soshnikov
2017-04-04

That's because block-level function declarations were standardized for backward compatibilities. So basically, they behave like function expressions in this case, and capture `{x: 50}` in their environment.

Same as:

```
var x = 10;
var foo;

with ({x: 50}) {
  foo = function() { return x; };
}

foo(); // 50
```

Xue Shihan
2017-04-16

Hi Dmitry,

Now I can figure out the value of this in most situation, but I feel confused when I see this:

```
var ViewModel = function() {
    this.clickCount = ko.observable(0);
    this.increment = function () {
```

```
        this.clickCount(this.clickCount() + 1);
    };
}

ko.applyBindings(new ViewModel());
```

The author uses Knockout. When I call `new ViewModel()`, this is set to the newly created object. If I call the method increment on the newly created object, `this.clickCount() + 1` is passed as an argument, I don't know how to decide the value of this when this appears in the function call.

Could you help me? Thank you.

Dmitry Soshnikov
2017-04-19

@Xue Shihan, if you call

```
<newlyCreatedObject>.increment();
```

then `this` is the `<newlyCreatedObject>`, and `this.clickCount()` is `<newlyCreatedObject>.clickCount()`.

Xue Shihan
2017-04-19

@Dmitry Soshnikov Thank you! Thank you for sharing your excellent knowledge with me.

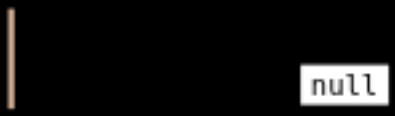
jason
2017-06-19

hello Dmitry, after reading the article several times, I am still confused with the case of a direct function calling such as "foo". I saw you answer

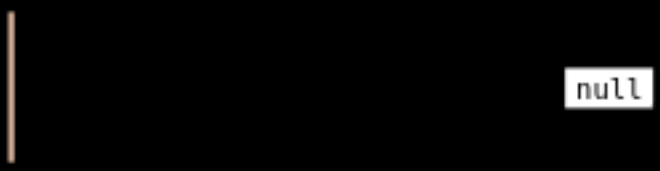
“foo() -> AO.foo() -> null.foo() -> foo.call(globalObject)”
But why is AO.foo() -> null.foo().
Dose the ecma specify that?

Dmitry Soshnikov
2017-06-20

@**jason**, correct, it’s the way it’s specified. In ES3 spec:



And later:



It is so in ES7+ spec.

Joe
2017-09-05

which version is ECMA-262-3? is it the 3rd edition (1999)?

Dmitry Soshnikov
2017-09-05

@**Joe**, correct, but the core features described in ES3, and ES5 are still the same in ES2017 today.

Shahnawaz Sharief
2017-10-13

Hi Dimitry,

I have a question – is every variable identifier in JS, a “Reference”, or of “Reference type”? And also, can the base value of a Reference type be a primitive type value such as a string or a number rather than an object?

Thanks

Dmitry Soshnikov
2017-10-18

@Shahnawaz Sharief

Starting ES5, yes, the component of a reference can be primitive value, see `HasPrimitiveBase` in ES5: 8.7.1, e.g.:

```
1 // base is primitive 'foo'
2 'foo'.toUpperString();
```

And yes, every variable once (i.e. accessed) from a code returns a value of the Reference type. The same with property access.

Arpit
2017-11-04

Hi Dmitry,
You have written some kickass article here, do appreciate a lot for the time and effort that you have put in and sharing it with everyone.

I am new to javascript, so, forgive me if my question seems silly, I am not able to grasp the value of “this” in two cases
1.)(foo.bar)(); // Reference, OK => foo

2.)(foo.bar = foo.bar)(); // global?

Would appreciate if you could explain it as to what exactly is happening there and I am not able to see anything in the following link
https://bclary.com/2004/11/07/#a-11.1.6

← Older Comments

Newer Comments →

Write a Comment

RELATED CONTENT BY TAG ECMA-262-3 ECMASCRIPT THIS