Dmitry Soshnikov in ECMAScript | 2010-04-10

ECMA-262-3 in detail. Chapter 8. Evaluation strategy

Read this article in: Russian, French.

> **Note:** this article was revisited and updated on **Feb 10th, 2020**

Introduction

In this small note we will consider strategy of passing arguments to functions in ECMAScript.

In general this part of computer science is known as *evaluation strategy*, that is, a set of rules for evaluating semantics of some *expressions*. The strategy of *passing arguments* to functions is one of its cases.

Many developers are used to think that objects in JavaScript are passed *by reference*, and the primitives — *by value*. In fact, this statement appears periodically in various articles, discussions, and even books on JavaScript. In this article we'll clarify, how valid this terminology is (and whether it's valid at all) when it comes to JavaScript.

General theory

Before diving into ECMAScript specifics, we need to discuss some parts of the general theory related to parameters passing.

Eager and Lazy evaluation

From the initial evaluation perspective, there are two main strategies: *strict* (sometimes it's called *eager*), meaning the arguments are evaluated *before* their application, and *non-strict*, meaning the evaluation happens on-demand, when the arguments are actually used (so-called *"lazy"* evaluation).

ECMAScript, as well as other languages (C, Java, Python, Ruby, etc) uses *eager* evaluation for the arguments:

```
function eager(x) {
  console.log(x);
}

eager(y); // ReferenceError: "y" is not defined
```

In the example above the error is thrown right and before the function call, that is, all arguments should be eagerly evaluated before passing.

The lazy evaluation can be achieved in JavaScript via callbacks:

```
function lazy(onDemandGetter) {
  console.log(onDemandGetter()); // ReferenceError: "y" is not defined
}

lazy(() => y); // OK
```

In the lazy evaluation the exception is throw only when we try to *access* th variable. In this case, the exception may be thrown, or *may not* — depen

ing on whether we actually need to use this argument, and whether the variable will exist in the future.

Arguments evaluation order

In addition, the *order* in which the arguments are evaluated and passed, is also standardized — it's *left-to-right*. Some other languages may use reverse evaluation order, that is, right-to-left. And in some languages it's not even specified, e.g. C++.

Example:

```
function foo(...args) {
  console.log(args); // [1, 2, 3]
}

foo(
  (() => { console.log(1); return 1; })(), // 1
  (() => { console.log(2); return 2; })(), // 2
  (() => { console.log(3); return 3; })(), // 3
);
```

Here we pass three arguments, and using the side-effect of logging each argument, we see that the left-to-right order is correctly maintained.

> **Note:** JavaScript uses **eager** evaluation of arguments, in **left-to-right** order.

Now let's talk about the ways of passing arguments to function. Since not all of the discussing below strategies are used in ECMAScript, in our examples we'll be using a *pseudo-code*, to describe and show the abstract algorithms.

Call by value

Let's start from the simplest strategy, known as *"by value"*.

This type of strategy is well-known by many developers. The value of an argument here is a *copy* of the passed object. The changes made inside the function *do not affect* the passed object outside. Technically, a runtime al

cates a *new memory block*, copies the *full contents* of the passing object into it, and then the function can use this new object from the new location.

```
// Pseudo-code:

bar = 10

procedure foo(barArg):
  barArg = 20;
end

foo(bar)

// changes inside foo didn't affect
// the bar which is outside
print(bar) // 10
```

For *primitive values* this strategy works just fine. However it can quickly become a performance bottleneck, when we have to deal with *larger complex structures*, that is with *actual objects*. And that's exactly what happens in C++ — a larger structure is *fully copied* to a new location when we pass it by value.

> **Note:** unless you explicitly need to, avoid passing a large object *by value* in C++. Use a *const reference* instead.

Let's create the two generic functions, which will be used to describe our following strategies. The first one, `modifyProperties`, just updates the properties of the object:

```
procedure modifyProperties(object):
  object.x = 100;
  object.y = 200;
end
```

The second one, `rewriteObject`, tries to fully *replace* the object contents with the new data, by *assigning* to the parameter:

```
procedure rewriteObject(object):
  object = {newX: 1, newY: 2};
end
```

And to reiterate the description of the *by-value* semantics, both functions *do not affect* our object:

```
point = {
  x: 10,
  y: 20
}

modifyProperties(point)

// Still the same:
print(point) // {x: 10, y: 20}

// Also, still the same:
rewriteObject(point) // {x: 10, y: 20}
```

Again, passing the full copy of an object may cause big performance issues, especially if we need to work with multiple such objects.

Now let's see what's the *by reference* strategy brings to the table.

Call by reference

In contrast, the *by reference* strategy receives *not a copy*, but the *implicit reference to an object* instead. And this reference is *directly* mapped (like an *alias*) to the object from the outside. Any changes to the parameter inside the function — either just the property modifications, or the full rewrite are *immediately reflected* on the original object.

Pseudo-code:

```
// Having the same point object:
point = {
  x: 10,
  y: 20
}

// Update the properties:
modifyProperties(point)

print(point) // OK, changed {x: 100, y: 200}

// Completely rewrite external contents:
rewriteObject(point)

print(point) // Yes, rewritten: {newX: 1, newY: 2}
```

That's said, passing by reference is much more efficient than passing by value. But in general theory (being just an alias) it allows full rewrite of the object contents. Now let's see at the *combined strategy*, known as *by-sharing*.

Call by sharing

If the first two strategies always were pretty familiar to developers, then the next strategy, and specifically its *terminology*, was not widely used. However as we will see shortly, exactly this strategy plays the key role in passing ECMAScript arguments.

Alternative names of this strategy are *"call by object"* or *"call by object-sharing"*.

Strategy *"by sharing"* was proposed and first named by Barbara Liskov for CLU programming language in 1974.

The main difference of this strategy is that function receives the *copy of the reference* to object. This reference copy is associated with the formal parameter and is its *value*.

Despite using the word *"reference"* in this description, this strategy *should not be confused with the "call by reference"* discussed above. The value of the argument is *not a direct alias*, but the *copy of the address*.

In this case a *re-assignment* to a new value *does not replace* the original object (as would *by reference* strategy do). However, since the formal parameter still received *an address*, it can access the *contents* (the properties) of the *original object*, and *mutate* them.

```
// Again the same point object:
point = {
  x: 10,
  y: 20
}

// By-sharing can update the properties:
modifyProperties(point)

print(point) // Yes, updated: {x: 100, y: 200}

// But it cannot fully rewrite its contents:
rewriteObject(point)

print(point) // Nope, still the: {x: 100, y: 200}
```

This strategy assumes that language in general operates with *objects* instead of the *primitive values*.

> Additional information on the semantics of this evaluation strategy can be found in the Name binding section of the Lexical Environments article. In particular, operations of Rebinding and Mutation allows to see the described process in detail.

By sharing as a special case of by value

Strategy *by sharing* is used in many languages: Java, ECMAScript, Python, Ruby, Visual Basic, etc.

Moreover, in Python community exactly this terminology — *by sharing* is used.

However, in other languages, e.g. in Java, ECMAScript, and others, this strategy is also called as *by value*, meaning *specific value — a reference copy*.

By sharing and pointers

Regarding C/C++, this strategy is similar to passing *by pointer*. Just in C it is still possible to *dereference* the pointer and *change the object from the outside*.

However, assigning a new value to a pointer just *rebinds* it to the new memory block, keeping the old memory untouched. Still it is possible to change properties of the original object using this pointer.

Therefore, making an analogy with pointers, we can obviously see that this is passing *by value of the address*, and what exactly pointer *is*. In this case, *by sharing* is some kind of "syntactic sugar" which at *assignment* behaves like a *"non-dereferening" pointer*, and in case of property changes — like a *reference*, not requiring the dereferencing operation. Sometimes it can be named as *"safe pointer"*.

C/C++ also has special "syntactic sugar" for this:

```
1 obj->x instead of (*obj).x
```

This idea can also be seen in C++ on the example of the `std::shared_ptr`. It also allows *sharing* an object between function arguments, and the outside world (that is, the function can modify the fields of the object), however the re-assignment only changes the pointer itself, and doesn't affect the outside object. This data type even called as *shared*_ptr.

Comparison table

Here is a small comparison table, which allows seeing the subtle difference between "by reference", and "by sharing" strategies.

Strategy	Value	Can modify contents?	Can replace contents?
By value	Full contents copy	No	No
By reference	Address copy	Yes	Yes
By sharing	Address copy	Yes	No

That's said, they differ only in the assignment semantics: "by reference" can replace the full contents, and "by sharing" instead rebinds the pointer to the new address.

In fact, *references* in C++ are just a *syntactic sugar* for pointers. At the lower level they are even compiled to the same exact instructions, and hold the same values — that is, addresses. However, references change the high-level semantics, making the assignment operator to behave differently then pointers, and which are using in "by sharing" strategy.

ECMAScript implementation

So now it is more clear, which evaluation strategy is used in ECMAScript — the *call by sharing*. That is, we can change the properties of the object, but cannot fully *re-assignment* it. Assignment only *rebinds* the parameter name to a new memory, keeping the original object *untouched*.

However as we mentioned earlier, the generic "by value" terminology for this strategy can be used among JS programmers — again, meaning the *value* of a pointer. JavaScript inventor Brendan Eich also notices that the copy of a reference is passed.

This behavior can also be seen on a *simple assignment*. In this case we can see that we have two *different variables*, but which *share* the *same value* — the *address* of the object.

ECMAScript code:

```
// Bind `foo` to {x: 10} object:
let foo = {x: 10};

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

// Bind `bar` to the *same* object
// as `foo` identifier is bound to:

let bar = foo;

console.log(foo === bar); // true
console.log(bar.x); // OK, also 10

// And now rebind `foo` to the new object:

foo = {x: 20};

console.log(foo.x); // 20

// And `bar` still points
// to the old object:

console.log(bar.x); // 10
console.log(foo === bar); // false
```

This *(Re)binding* operation (that is, setting the *variable value* to an object address) can be illustrated on the following figure:

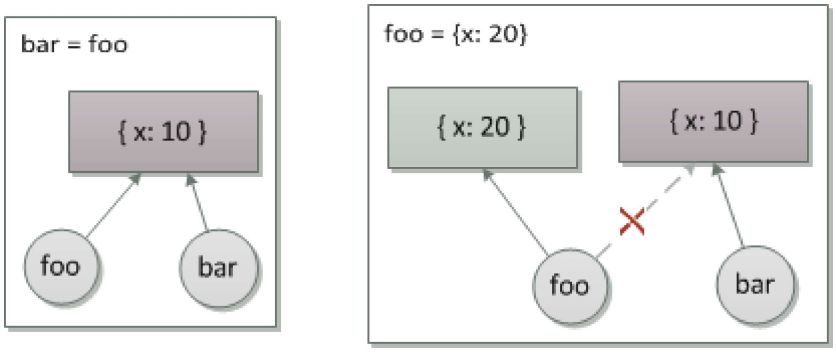

Figure 1. Rebinding.

Assignment of one variable to another just *copies* its address, making both variables *pointing* to the *same memory location*. The following assignment of a new value *unbinds* a name from the old address and *rebinds* it to the new one. This is an important difference from the *by reference* strategy, a this is exactly how objects are also passed as arguments to functions.

And once we have an address of an object, we can mutate its contents (updated properties) — and this is operation of *Mutation*.

Terminology versions

Let's define the versions of correct terminology related to ECMAScript in this case.

It can be either *"call by value"*, with specifying that the *special case of call by value is meant* — when the value is the *address copy*. From this position it is possible to say that *everything in ECMAScript is passed by value*.

Or, *"call by sharing"*, which makes this distinction from "by reference", and "by value". In this case it is possible to separate passing types: *primitive values are passed by value and objects — by sharing*.

The statement *"objects are passed by reference"* formally *is not related to ECMAScript* and is incorrect.

Conclusion

I hope this article helped to understand in more details evaluation strategy in general, and also related to ECMAScript. As always, I'll be glad to discuss and answer the questions in the comments.

Additional literature

External articles:

- Evaluation strategy
- Call by value
- Call by reference
- Call by sharing

ECMA-262-5 in detail:

- Name binding section of the Lexical Environments chapter

Translated by: Dmitry A. Soshnikov
Published on: 2010-04-10
Revisited and updated on: 2020-02-10

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

Originally published on: 2009-08-11

August 2014
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

Meta

Log in

Dmitry Soshnikov

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

Published

2010-04-10

Write a Comment

23 COMMENTS

Robert Polovsky

2010-04-11

Thank you Dmitry, very informing article.

denisdeng

2010-04-16

thank you!

Yonatan

2010-05-02

gr8 ending to a gr8 serie of articles, thanks! (will reread 6 articles again... not including the OOP articles :P)

Morten Krogh

2011-01-23

Hi Dmitry

Very good Javascript series. Excellent. I have read several of your articles, and they are really good.

One question about this article. Are there languages that use call by reference? You don't give any examples. The closest I could think of is in C by using pointers to pointers.

```
1  int f(struct example **x)
2  {
3    *x = malloc(sizeof **x);
4    ......
5  }
```

or you could wrap a struct inside a struct.

But are there languages with more direct examples, where f(x) would really be by reference?

Morten.

Dmitry A. Soshnikov

2011-01-24

@Morten Krogh

> *Are there languages that use call by reference?*

First, we should understand that mentioned strategies are just *abstract* descriptions.

However, in some languages, the concept and the exact meaning of a *reference* can vary. The nearest example of *by-reference* strategy can be shown with C++ and its reference sugar:

```
1  // C++
2
3  void call_by_reference(int& x)
4  {
5    x = 20; // external *mutation*
6  }
7
8  void call_by_pointer(int* x)
9  {
10   *x = 20; // also external *mutation*
11   x = new int; // and this is already *rebinding*
12   *x = 30; // mutation of the other memory block
13 }
14
15 void call_by_value(int x)
16 {
17   x = 20; // just local mutation
18 }
19
20 int _tmain(int argc, _TCHAR* argv[])
21 {
22   int y = 0;
23
24   call_by_reference(y);
25   call_by_pointer(&y);
26   call_by_value(y);
27
28   return 0;
29 }
```

Thus, as said, this *by-reference* of C++ is just a syntactic sugar (in general, and if to see the assembly generated code) for the *by-pointer* (which is described above *by-sharing*) strategy — i.e. for not to dereference a pointer every time.

The main point is in the operation of *assignment*. Because often enough there are questions on JS such as "Why if an object is passed by reference I can't replace it via assignment?". The answer is — it's not by-reference (from C++ viewpoint), but "by-pointer" (or "by-value, where value is an address"), where assignment (without dereferencing) just changes the pointer's address.

Also take a look on this explanation of a name binding concept, which also accurately explains the difference. Thus, the assignment means a rebinding. And changing of a properties means mutation.

Dmitry.

Morten Krogh

2011-01-25

Hi Dmitry

Yes, but do you know a language where all function arguments are by reference. So for example,

```
function f(x) {
  x = 5
};

a = 10;
f(a);
```

and now `a = 5`.

It would be a weird language to work with.

Morten.

Dmitry A. Soshnikov
2011-01-25

@Morten Krogh

Nope, I'm not aware about such a language. Seems, it wouldn't be very useful 😉

Dmitry.

eric
2011-09-23

I'm not completed assent this article. since in the java world, pass value to method. and if do like you said, pass value will not affect outside object. But its definite will change the properties of outside object, because the parameter variable hold the reference of the original object, if don't change the reference, change any properties will affect outside original object(this strategy more like your article "call by reference", but it is called pass by value).

eric
2011-09-23

And i know in the C# there has two keyword–"ref"|"out", this can do like you said "call by share"

eric
2011-09-23

Strategy by sharing is used in many languages: Java, ECMAScript, Python, Ruby, Visual Basic, etc.

Java can call by sharing?I doubt...

http://www.javaworld.com/javaworld/javaqa/2000-05/03-qa-0526-pass.html

eric

2011-09-23

> *Hi Dmitry*
>
> *Yes, but do you know a language where all function arguments are by reference. So for example,*
>
> ```
> function f(x) {
>   x = 5
> };
>
> a = 10;
> f(a);
> and now a = 5.
> ```
>
> *It would be a weird language to work with.*
>
> *Morten.*

C# can do this

```
function f(ref x) {
   x = 5;
}
a = 10;
f(a);
```

`a` will be the 5

Dmitry A. Soshnikov

2011-09-24

@eric

> *in the java world, pass value to method ... (this strategy more like your c ticle "call by reference", but it is called pass by value)*

Yes, in Java the terminology “by-value” is used. But the semantics is what is described in this article as “by-sharing”. As noticed above in the last paragraph terminology versions, both terminologies can be used: either “by-sharing” or “by-value when the value is the reference”.

eric

2011-09-25

Thanks for your reply. I’m not read very careful, get mixed up “call by reference” and “call by sharing”. In your article, “call by reference” means pass the real object to method, and “call by sharing” ,means pass origin object address copy to method. really need more carefully.

M

2012-05-20

I just use the term “call-by-value” for this — whether the value being passed is the address of an object, or whether it’s an integer, it doesn’t matter. The run-time just copies the bit pattern into the local variable of the called routine. No decision making is needed by the run-time. Note that I don’t actually *know* how the run-time executor is written. But there’s just no need to do anything else.

As far as the question whether any language allows the calling value to be changed, yeah, several (mostly archaic) languages do, such as Pascal. That concept *is* call-by-reference, and here’s the syntax in Pascal:

```
procedure f(var arg1: integer);
    begin
        arg1 := 5;
    end;
...
i = 10;
f(i);
// i now holds 5!
```

The “var” key work means call-by-reference. Memory for arg1 is not allocated because it’s using the calling routine’s memory location.

Dmitry Soshnikov
2012-05-25

@M

Yes, absolutely correct, if you treat as the “integer” address is passed, then you may name it “by-value” which is correct. Notice though, that runtime nevertheless should distinguish form “simple values” and “address values.

Beside Pascal, C++ (as a sugar for dereferenced pointer) or PHP has references as well.

lail3344
2012-11-26

Thanks Dmitry.
I read all of these articles, they are very helpful and clear for whom learning java-script like me.

hsd
2014-09-24

good job

Francesco
2017-02-06

Great article! Thank you!

Ref vs val

2018-01-24

Hey Dmitry.
Can you clarify me something?
(by value)

```
var a = 4;
var b = 6;
```

means:
(address 0x00) var a has value of 4
(address 0x04) var a has value of 6

(by refference)

```
var a = {a: 4};
var b = a;
```

means:
(address of object 0xdd) has few things
(address 0x00) var a has value of 0xdd
(address 0x04) var a has value of 0x00 or 0xdd?

Dmitry Soshnikov

2018-01-26

@**Ref vs val**,

In JS this is abstracted in the environments concept. So really can just be presented as:

```
const env = {
  a: 4,
  b: 6,
};
```

```
5
6   env.a = {a: 4};
7   env.b = env.a;
```

The “memory” references are abstracted away with the environments model — you may think that `env.a` in first case holds the *value* `4` *directly* (not a reference to value 4). Whereas in `env.a = {a: 4}`, it holds a *reference* to the object allocated “somewhere else” (on the heap).

In general (irregardless JS), an environment (symbol table) maps identifiers to memory locations, yes. So in this case even in the first case the `env.a` would hold a *memory address*, which stores value `4`. And in the second case it would hold a memory address, at which a *pointer to the object* is stored.

Murad Sofiyev

2019-11-10

Actually I don’t understand this sentence “This strategy assumes that language in the majority operates with objects, rather than primitive values.”
What you mean exactly in this sentence? For explaining this sentene can you give me some examples?

Ocup Huang

2021-07-18

Thank you for sharing this 🙂

But I think there’s a tittle bug in the code at the ECMAScript implementation part that it can not be rebound to the “const” foo.

// Bind `foo` to {x: 10} object:
const foo = {x: 10};
// And now rebind `foo` to the new object:
foo = {x: 20};

Dmitry Soshnikov

2021-07-20

@**Ocup Huang**, thanks, fixed the typo!

Write a Comment

RELATED CONTENT BY TAG BY REFERENCE BY SHARING BY VALUE ECMA-262-3 ECMASCRIPT EVALUATION STRATEGY