David Shariff

MANAGE THE UI PLATFORM TEAM @ AMAZON.COM

Follow @davidshariff 2,077 followers

What is the Execution Context & Stack in JavaScript?

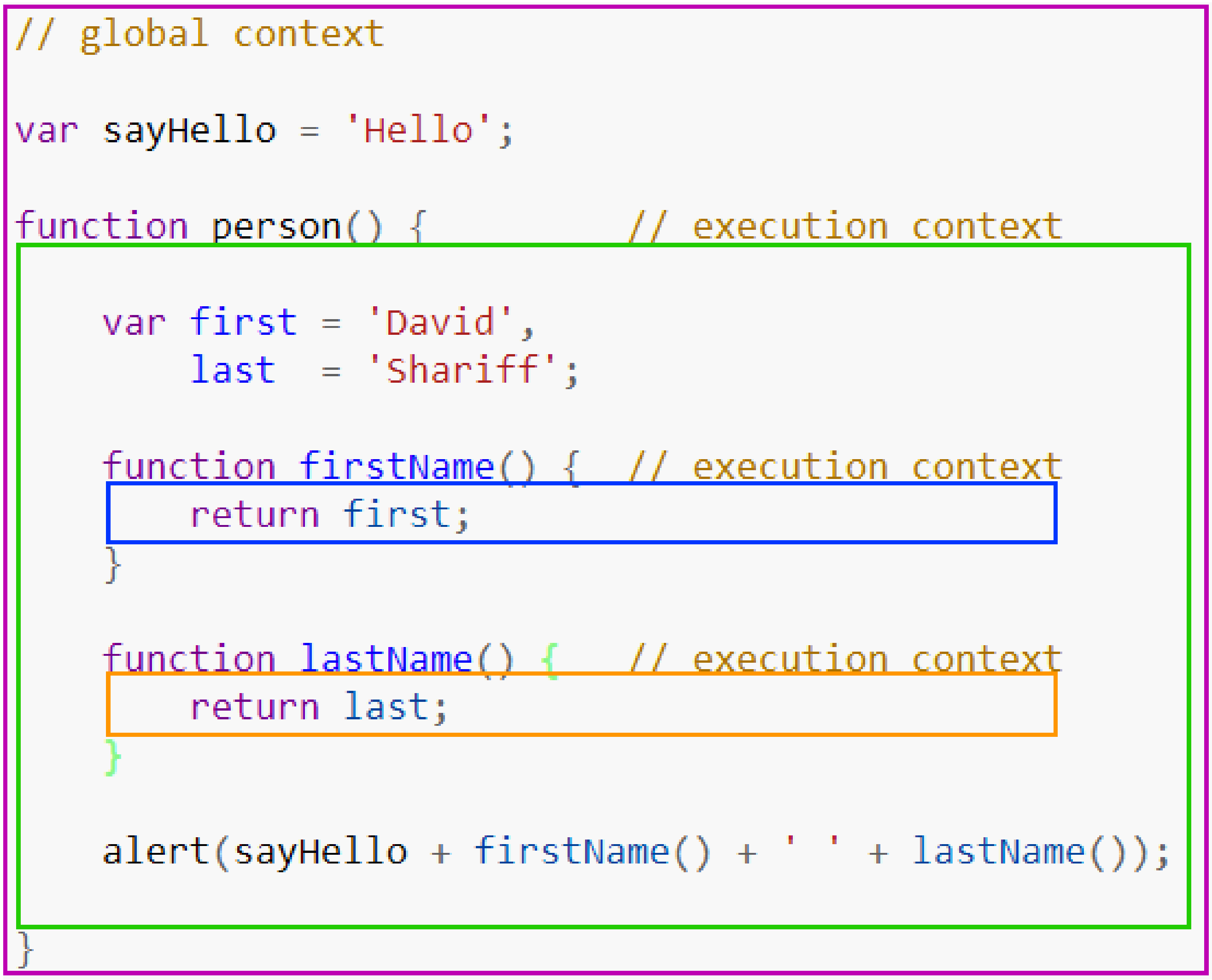
In this post I will take an in-depth look at one of the most fundamental parts of JavaScript, the `Execution Context`. By the end of this post, you should have a clearer understanding about what the interpreter is trying to do, why some functions / variables can be used before they are declared and how their value is really determined.

What is the Execution Context?

When code is run in JavaScript, the environment in which it is executed is very important, and is evaluated as 1 of the following:

- **Global code** – The default envionment where your code is executed for the first time.
- **Function code** – Whenever the flow of execution enters a function body.
- **Eval code** – Text to be executed inside the internal eval function.

You can read a lot of resources online that refer to `scope`, and for the purpose of this article to make things easier to understand, let's think of the term `execution context` as the envionment / scope the current code is being evaluated in. Now, enough talking, let's see an example that includes both `global` and `function / local` context evaluated code.



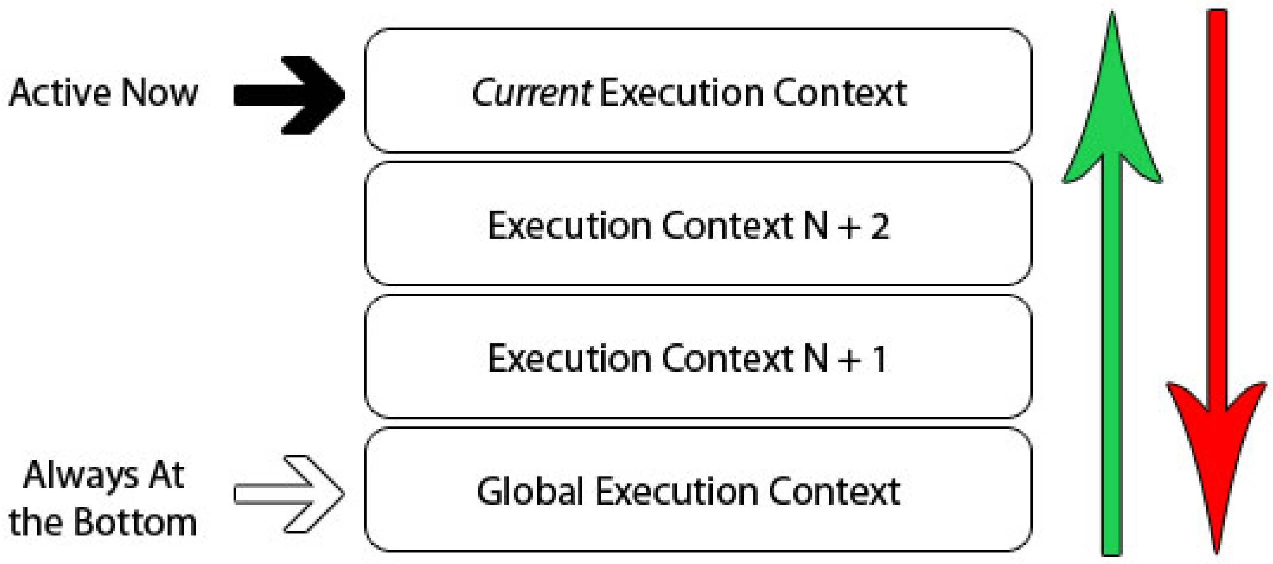
```
// global context

var sayHello = 'Hello';

function person() {         // execution context

    var first = 'David',
        last  = 'Shariff';

    function firstName() {  // execution context
        return first;
    }

    function lastName() {   // execution context
        return last;
    }

    alert(sayHello + firstName() + ' ' + lastName());

}
```

Nothing special is going on here, we have 1 `global context` represented by the purple border and 3 different `function contexts` represented by the green, blue and orange borders. There can only ever be 1 `global context`, which can be accessed from any other context in your program.

You can have any number of `function contexts`, and each function call creates a new context, which creates a private scope where anything declared inside of the function can not be directly accessed from outside the current function scope. In the example above, a function can access a variable declared outside of its current context, but an outside context can not access the variables / functions declared inside. Why does this happen? How exactly is this code evaluated?

Execution Context Stack

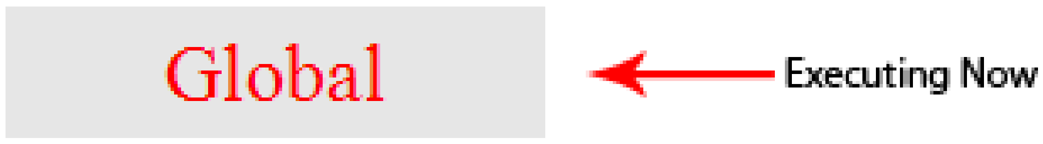
The JavaScript interpreter in a browser is implemented as a single thread. What this actually means is that only 1 thing can ever happen at one time in the browser, with other actions or events being queued in what is called the `Execution Stack`. The diagram below is an abstract view of a single threaded stack:



As we already know, when a browser first loads your script, it enters the `global execution context` by default. If, in your global code you call a function, the sequence flow of your program enters the function being called, creating a new `execution context` and pushing that context to the top of the `execution stack`.

If you call another function inside this current function, the same thing happens. The execution flow of code enters the inner function, which creates a new `execution context` that is pushed to the top of the existing stack. The browser will always execute the current `execution context` that sits on top of the stack, and once the function completes executing the current `execution context`, it will be popped off the top of the stack, returning control to the context below in the current stack. The example below shows a recursive function and the program's `execution stack`:

```
(function foo(i) {
    if (i === 3) {
        return;
    }
    else {
        foo(++i);
    }
}(0));
```



The code simply calls itself 3 times, incrementing the value of i by 1. Each time the function `foo` is called, a new execution context is created. Once a context has finished executing, it pops off the stack and control returns to the context below it until the `global context` is reached again.

There are 5 key points to remember about the `execution stack`:

- Single threaded.
- Synchronous execution.
- 1 Global context.
- Infinite function contexts.
- Each function call creates a new `execution context`, even a call to itself.

Execution Context in Detail

So we now know that everytime a function is called, a new `execution context` is created. However, inside the JavaScript interpreter, every call to an `execution context` has 2 stages:

1. **Creation Stage** [when the function is called, but before it executes any code inside]:
    - Create the Scope Chain.

- Create variables, functions and arguments.
- Determine the value of `"this"`.

2. **Activation / Code Execution Stage**:
    - Assign values, references to functions and interpret / execute code.

It is possible to represent each `execution context` conceptually as an object with 3 properties:

```
executionContextObj = {
    'scopeChain': { /* variableObject + all parent execution context's varia
    'variableObject': { /* function arguments / parameters, inner variable a
    'this': {}
}
```

Activation / Variable Object [AO/VO]

This `executionContextObj` is created when the function is invoked, but *before* the actual function has been executed. This is known as stage 1, the `Creation Stage`. Here, the interpreter creates the `executionContextObj` by scanning the function for parameters or arguments passed in, local function declarations and local variable declarations. The result of this scan becomes the `variableObject` in the `executionContextObj`.

Here is a pseudo-overview of how the interpreter evaluates the code:

1. Find some code to invoke a function.
2. Before executing the `function` code, create the `execution context`.
3. Enter the creation stage:
    - Initialize the `Scope Chain`.
    - Create the `variable object`:
        - Create the `arguments object`, check the context for parameters, initialize the name and value and create a reference copy.
        - Scan the context for function declarations:
            - For each function found, create a property in the `variable object` that is the exact function name, which has a reference pointer to the function in memory.
            - If the function name exists already, the reference pointer value will be overwritten.
        - Scan the context for variable declarations:

    - For each variable declaration found, create a property in the `variable object` that is the variable name, and initialize the value as undefined.
    - If the variable name already exists in the `variable object`, do nothing and continue scanning.
  - Determine the value of `"this"` inside the context.

4. Activation / Code Execution Stage:
   - Run / interpret the function code in the context and assign variable values as the code is executed line by line.

Let's look at an example:

```
function foo(i) {
    var a = 'hello';
    var b = function privateB() {

    };
    function c() {

    }
}

foo(22);
```

On calling `foo(22)`, the `creation stage` looks as follows:

```
fooExecutionContext = {
    scopeChain: { ... },
    variableObject: {
        arguments: {
            0: 22,
            length: 1
        },
        i: 22,
        c: pointer to function c()
        a: undefined,
        b: undefined
    },
    this: { ... }
}
```

As you can see, the `creation stage` handles defining the names of the properties, not assigning a value to them, with the exception of formal arguments / parameters. Once the `creation stage` has finished, the flow of execution enters the function and the activation / code `execution stage` looks like this after the function has finished execution:

```
fooExecutionContext = {
    scopeChain: { ... },
    variableObject: {
        arguments: {
            0: 22,
            length: 1
        },
        i: 22,
        c: pointer to function c()
        a: 'hello',
        b: pointer to function privateB()
    },
    this: { ... }
}
```

A Word On Hoisting

You can find many resources online defining the term `hoisting` in JavaScript, explaining that variable and function declarations are *hoisted* to the top of their function scope. However, none explain in detail why this happens, and armed with your new knowledge about how the interpreter creates the `activation object`, it is easy to see why. Take the following code example:

```
(function() {

    console.log(typeof foo); // function pointer
    console.log(typeof bar); // undefined

    var foo = 'hello',
        bar = function() {
            return 'world';
        };

    function foo() {
        return 'hello';
    }

}());
```

The questions we can now answer are:

- **Why can we access foo before we have declared it?**
    - If we follow the `creation stage`, we know the variables have already been created before the `activation / code execution stage`. So as the function flow started executing, `foo` had already been defined in the `activation object`.
- **Foo is declared twice, why is foo shown to be `function` and not `undefined` or `string`?**

  - Even though `foo` is declared twice, we know from the `creation stage` that functions are created on the `activation object` before variables, and if the property name already exists on the `activation object`, we simply bypass the decleration.
  - Therefore, a reference to `function foo()` is first created on the `activation object`, and when the interpreter gets to `var foo`, we already see the property name `foo` exists so the code does nothing and proceeds.
- **Why is bar `undefined`?**
  - `bar` is actually a variable that has a function assignment, and we know the variables are created in the `creation stage` but they are initialized with the value of `undefined`.

Summary

Hopefully by now you have a good grasp about how the JavaScript interpreter is evaluating your code. Understanding the execution context and stack allows you to know the reasons behind why your code is evaluating to different values that you had not initially expected.

Do you think knowing the inner workings of the interpreter is too much overhead or a necessity to your JavaScript knowledge ? Does knowing the execution context phase help you write better JavaScript ?

Note: Some people have been asking about closures, callbacks, timeout etc which I will cover in the next post, focusing more on the Scope Chain in relation to the `execution context`.

Further Reading

- ECMA-262 5th Edition
- ECMA-262-3 in detail. Chapter 2. Variable object
- Identifier Resolution, Execution Contexts and scope chains

Related Posts...

1 **Identifier Resolution and Closures in the JavaScript Scope Chain**

2 **JavaScript Inheritance Patterns**

3 **Futures and Promises in JavaScript**

4 **JavaScript's 'this' Keyword**

5 **Chaining Variable Assignments in JavaScript: Words of Caution**

Comments

Joe *said* on 21/06/2012 at 6:25 pm:

I'd love to read this, but that sidebar is a deal breaker!

Reply ↓

David Shariff *said* on 23/06/2012 at 6:55 pm:

@Joe
Noted. I made a few changes already to the nav and social share.

Reply ↓

Jon *said* on 21/06/2012 at 10:28 pm:

Awesome little article, good explanation of functions scope. I'm definitely bookmarking this.

Reply ↓

Les *said* on 26/06/2012 at 4:07 pm:

Great Article! Def cleared up a few gaps in my understanding.

Reply ↓

Loke *said* on 29/06/2012 at 11:21 pm:

Great article! I would recommend this to anyone looking to improve JS performance in their code.

Reply ↓

Dave Stibrany *said* on 30/06/2012 at 3:06 am:

Ahhhh, ok now I can actually conceptualize what hoisting is really doing, on an actual code level basis.

Thanks for this, looking forward to your scope chain article!

Reply ↓

Wojtek Urbanski ***said*** **on 30/06/2012 at 3:48 pm:**

Thanks for this great article! I was aware of 90% of this before. Now I know what is the logic behind and it makes a lot more sense. Thanks, I'm waiting for more!

Reply ↓

Muhammad Atif ***said*** **on 01/07/2012 at 5:21 pm:**

Hi david.

Great article. But I couldn't get one thing. According to your executionContextObj.

```
fooExecutionContext = {
    variableObject: {
        arguments: {
            0: 22,
            length: 1
        },
        i: 22, ........
```

arguments and i is stored separately. so they shouldnt interfere with each other. but reality is when within function I update arguments[0] it changes 'i' or when I change 'i' it changes arguments[0] as well. Would you like to explain?

Thank you,
Atif, Pakistan.

Reply ↓

David Shariff ***said*** **on 20/07/2012 at 1:30 pm:**

@Muhammad Atif
Although the arguments and formal parameter names are in separate places in the VO, they are actually a reference to each other, hence why when you change 1 the other changes.

Reply ↓

Cosmo *said* on 09/07/2016 at 4:53 am:

What does "they are actually a reference to each other" mean here? If the argument is an object, I can understand both i and arguments[0] are holding the same reference to this object. But what about primitive value? I don't think we can doing something like holding a reference to a primitive since we usually just copy its value.

Reply ↓

Rajnish Kumar *said* on 02/07/2012 at 1:31 pm:

Wonderful article.Really loved reading this article.

Reply ↓

John Weis *said* on 04/07/2012 at 4:31 am:

Very thorough. Loved it.

Reply ↓

Chris Webb *said* on 09/07/2012 at 3:20 pm:

Great to see this broken down step by step with visuals. Thanks for taking the time to share knowledge!

Reply ↓

fox *said* on 13/07/2012 at 11:27 am:

great article, thank u

Reply ↓

marveltracker *said* on 05/08/2012 at 12:25 pm:

Very helpful post. Thanks

Reply ↓

Jayasankar *said* on 15/09/2012 at 4:23 pm:

Thanks David, very nice article very useful defiantly cleared few doubts please post more articles when you get time.

Reply ↓

Sunny *said* on 10/10/2012 at 12:09 am:

Really Good article. Keep it up!

Reply ↓

Gaurav *said* on 11/10/2012 at 3:13 pm:

Thanks for this great post. It’s fantabulous.

Reply ↓

Rizwan *said* on 15/11/2012 at 1:46 pm:

@Muhammad Atif,
And remember this dynamic relationship between arguments object and formal parameters exists only for the parameters which get value during function call. e.g function test(a,b){} is called as test(1) then changing arguments[0] will change value of a and vice-versa but changing arguments[1] will have no effect on value of b.

Reply ↓

Natasha *said* on 03/01/2017 at 12:19 pm:

Hi, Rizwan!
Please, give me an example of the function test(a,b){} and it’s invocation and I’ll try to show you that it is possible. Thanks!

Reply ↓

Natasha *said* on 04/01/2017 at 3:23 pm:

Sorry. I was wrong.
Regards,
Natasha.

Reply ↓

Andrew Hedges *said* on 28/11/2012 at 8:50 am:

Here's a little JavaScript puzzle that illustrates the point about the arguments object and variable declarations.

Reply ↓

jasonzhuang *said* on 06/01/2013 at 7:28 pm:

@David Shariff, Could you explain the execution context for the following code:

```
Callbacks = function() {
    var fire = function(context, args){
          console.log("this is private fire() function");
        },
        self = {
            fireWith:function(context,args){
                console.log("this is self.fireWith()");
                fire(context, args);//mark line
                return this;
            },
            fire:function(){
                console.log("this is self.fire()");
                self.fireWith(this, arguments);
                return this;
            }
        };
        return self;
    }
    var call = new Callbacks();
    call.fire();
}
```

Why fire method in self.fireWith() not call self.fire()?

Reply ↓

ManInHat *said* on 17/07/2014 at 11:35 pm:

I'll take a stab at this, although it's been over a month:

1. In your code, fire() in the "self" object doesn't reference an object method (as in "self.fire" or "this.fire")
2. "fire()" is not defined within the context of the self.fireWith() function; it is declared as a variable alongside "self".

As a result, calling "fire()" pulls in (var fire). Hope this helps!

Reply ↓

Natasha *said* on 03/01/2017 at 3:45 am:

1) This code contents some mistakes. Without of them it looks like this:

```
var Callbacks = function() {
    var fire = function(context, args){
          console.log("this is private fire() function");
        },
        self = {
            fireWith:function(context,args){
                console.log("this is self.fireWith()");
                fire(context, args);//mark line
                return this;
            },
            fire:function(){
                console.log("this is self.fire()");
                self.fireWith(this, arguments);
                return this;
            }
        };
        return self;
    };
    var call = new Callbacks();
    call.fire();
```

2) Because this constructor Callbacks (this name is not suitable for a constructor) returns an object (self) the variable call points to self and call.fire() is the same as self.fire().
3) the first statment of self.fire outputs the string “this is self.fire()”.
4) the next statment of self.fire invokes self.fireWith(this, arguments).
5) the first statment of self.fireWith(this, arguments) outputs the string “this is self.fireWith()”.
6) the next statment of self.fireWith(this, arguments) invokes (!!!) the function fire(context, args), NOT the method of the object self self.fire().
I hope my exolanation is clear.
Thanks!

Reply ↓

Max Yan *said* on 07/01/2013 at 1:10 am:

Thanks for sharing.really help

Reply ↓

Marvin *said* on 28/01/2013 at 5:12 pm:

Great post explained in layman terms.

Reply ↓

Torro *said* on 31/01/2013 at 10:47 am:

Great use of images to explain. Really enjoyed it.

Reply ↓

miro *said* on 26/02/2013 at 12:05 am:

Great article, real in depth look at functions behind the scene. Exactly what I needed. Thank you.

Reply ↓

Ashish Krishan *said* on 26/03/2013 at 4:35 am:

Good article. Explained very well

Reply ↓

Willson *said* on 30/03/2013 at 2:16 am:

Thanks for the great explanation here! I feel the best way to explain JS scope is visually, and your article really articulates this very well.

Reply ↓

GuyK *said* on 03/04/2013 at 4:59 pm:

Great article, clear explanation.
Thanks!

Reply ↓

Maizere *said* on 05/07/2013 at 2:28 am:

Great Article ,simply great.Also many thanks to David for this wonderful article. Thanks Again, I’m waiting for more! heart touching articles.Cleared up every confusions.

Reply ↓

Sameer Pathak(Nepali) ***said*** **on 06/07/2013 at 8:52 pm:**

Perfect artical for those even with poor english.

Reply ↓

pepecristiano ***said*** **on 12/07/2013 at 4:26 am:**

What happens in the CA when among the arguments you have a call-back anonymous function?

Reply ↓

theWebStoreByG! ***said*** **on 29/07/2013 at 11:51 am:**

First time reader… Brilliant article

Now I definitely have a better understanding of the execution context.

Reply ↓

raziq *said* on 02/10/2013 at 3:27 pm:

can u please make tutorial on function's [[scope]] property how does it work.

Reply ↓

David Shariff *said* on 02/10/2013 at 3:31 pm:

Raziq

You can try reading my article on Scope Chain.

Reply ↓

raziq *said* on 02/10/2013 at 5:37 pm:

i am currently reading " javascript professional for web developer 3rd edition " book there it talks about functions internal [[scope]] property which i dont understand very well i thought you can explain it better, and yes i have read your article on scope chain its very good and well written but there is no mention of function's [[scope]] property

Reply ↓

mani *said* on 24/10/2013 at 6:06 pm:

Really very nive article, it clears my doubts on how javascript interpreter works

Reply ↓

RobG *said* on 13/12/2013 at 11:19 am:

Good article, but note that when entering function code, the ThisBinding is set first, not last. See ECMA-262 §10.4.3 (http://ecma-international.org/ecma-262/5.1/#sec-10.4.3)

Reply ↓

David Shariff *said* on 22/01/2014 at 4:30 am:

This article is referring to ECMA:262-3 Section 10.2.3

"When control enters an execution context, the scope chain is created and initialised, variable instantiation is performed, and the this value is determined."

Reply ↓

---

md khan ***said*** **on 14/01/2014 at 11:57 pm:**

does variable is hoisted first or function?

```
(function foo(){
  return boo;
  function boo(){return this;}
  var boo = 5;
})()



or
(function foo(){
  return boo;
    var boo = 5;
  function boo(){return this;}


})()
```

both returns (in chrome),

function boo(){return this;}

Reply ↓

David Shariff ***said*** **on 22/01/2014 at 4:40 am:**

Functions are hoisted before variables.

Reply ↓

Khushi ***said*** **on 24/01/2014 at 7:27 pm:**

Great Article. Loved it.

Reply ↓

Sriram *said* on 04/02/2014 at 5:40 pm:

Awesome article. Got a detailed understanding on many concepts which haunted me for long time. Great work dude..

Reply ↓

Yufeng *said* on 28/03/2014 at 2:18 am:

Really helpful for understanding the fundamental concept of the execution context. Great work !

Reply ↓

Andy Gray *said* on 08/04/2014 at 7:22 am:

Superb explanation. A model of clarity. When is the book available?

Reply ↓

Vishwa *said* on 12/04/2014 at 1:28 pm:

Good explanation

Reply ↓

Vishwa *said* on 12/04/2014 at 2:45 pm:

Hi David,

would u please explain why last foo type is string? am I misunderstanding something

```
console.log(typeof foo); // function pointer
    var foo = 'hello';
    console.log(typeof foo); // string
    function foo() {
        alert(1);
    };
    console.log(typeof foo); // string
```

Reply ↓

rp *said* on 16/11/2017 at 1:18 pm:

because the Functions is hoisted,so function declation is useless in code executing

Reply ↓

hermit8888 ***said*** **on 25/09/2014 at 7:35 am:**

I was reading the section on execution context in “Professional JavaScript for Web Developers” and was left somewhat confused, your explanation here cleared all that up, thank you!

Reply ↓

Kevin ***said*** **on 07/10/2014 at 2:49 pm:**

Hi David,

I enjoyed reading your post on execution context.
However, I have a doubt when I tried executing the following code:

```
function abc()
{
  var a = 1;
  pqr();
}
function pqr()
{
  alert(a);
}
abc();
```

Once I fire this code, it says that the variable “a” is not defined.

I wrote this code only after I read that, “So we now know that everytime a function is called, a new execution context is created.”

In the above case, the variable “a” was a part of function abc()
From abc() i called pqr() which I think created a new context and so the control went from abc() to pqr();

Now my doubt is that since pqr() is on the top of the execution stack, it should logically be able access variable “a” that is declared inside abc() because of the scope chain.

Can you please explain why my code did not run?

Awaiting your reply

Reply ↓

David Shariff ***said*** **on 15/12/2014 at 4:49 am:**

Each execution context has its own VO. JavaScript scoping is lexical, not dynamic. You can read more on scoping here http://davidshariff.com/blog/javascript-scope-chain-and-closures/

Reply ↓

Chunming Cao ***said*** **on 24/02/2016 at 4:21 am:**

The scope chain is not related with the execution stack.

The scope chain is lexical, and it is established when the funciton is created.

You can call function pqr() multiple times. Wherever you call pqr(), the scope chain of pqr is the same. and the 'a' inside of pqr always refer to the globle variable.

Reply ↓

Agradip ***said*** **on 08/12/2014 at 6:09 pm:**

THANKS A LOT ,AWESOME ARTICLE

Thanks,
Agradip

Reply ↓

Jagadish Dharanikota ***said*** **on 12/12/2014 at 6:43 pm:**

Really like the article David. Every frontend developer should read this article. Thank you very much.

Reply ↓

Pablo Perez ***said*** **on 14/02/2015 at 12:31 pm:**

Thank you for the aritlce David…great read

Reply ↓

Mohamed *said* on 17/07/2015 at 5:16 am:

Thanks a lot for this article, I was struggling against these concepts, but I have a better understanding now thanks to you.

Reply ↓

Skyler *said* on 22/10/2015 at 11:43 pm:

Amazingly helpful explanation! Thank you so much. Would you happen to have to any explanations like this regarding closure? Also, what sorts of aspects are covered in your book? Thank you again!

Reply ↓

angel calderaro *said* on 23/10/2015 at 11:08 am:

Really like the article!

Reply ↓

Ankit Mongia *said* on 03/01/2016 at 8:24 pm:

gr8 article.

Reply ↓

Gourav Batra *said* on 29/02/2016 at 6:46 pm:

Awesome Article For Javascript Internals…….

Thanks

Reply ↓

Aung Gyan *said* on 16/03/2016 at 11:55 am:

Hi David,

I’m now reading You Don’t Know JS book. According to writer, Javascript also compiles the code before executes it. Can we say Creation Stage as Compilation Stage?

Reply ↓

monica *said* on 23/03/2016 at 10:30 pm:

Great article! Very good explanation! I wish there were more articles like this. I've just been added to the Smart Web Restaurant's team and it really helps me a lot to make everything clear with the team's code. Thank you, David! Not everyone can write an article in 2012 that will be still relevant in 2016.

Reply ↓

luo *said* on 12/04/2016 at 2:25 pm:

Hi David,

"when the function is called, but before it executes any code inside"

means :
function test(){
var a =1;
}

or
test()?

Reply ↓

Anon *said* on 16/08/2017 at 3:09 am:

A) "function foo() {}" is a function declaration.

B) "foo()" is a function invocation.

The creation stage starts when you call (invoke) the function (line B). The code defined in the functions body during its declaration (line A) will only be executed during the code execution stage.

Since the creation stage happens before the code executing stage, the statement is right. The creation stage starts "when the function is called" (line B) "but before it executes any code inside", which will happen during the execution stage.

Reply ↓

dharmu *said* on 24/07/2016 at 3:48 pm:

Great article.

I am not good at javascript and I am bit confused with below example
I have added two console logs

```
(function() {
    console.log(typeof foo); // function pointer
    console.log(typeof bar); // undefined



    var foo = 'hello',
        bar = function() {
            return 'world';
        };


    console.log(typeof foo); // added
    function foo() {
        return 'hello';
    }
    console.log(typeof foo); // added
}());
```

Output is:
function
undefined
string
string

I expected that it should have been
function
undefined
function
function

or

function
undefined
string
function

Reply ↓

Anon *said* on 16/08/2017 at 3:01 am:

So many people asking the same question… Check my replies to Thomas and Manx, below.

Reply ↓

Thomas *said* on 02/08/2016 at 5:35 am:

```
(function() {


    console.log(typeof foo); // function pointer
    console.log(typeof bar); // undefined
    var foo = 'hello',
        bar = function() {
            return 'world';
        };


    function foo() {
        return 'hello';
    }
    console.log(foo); // string
    foo(); //  Uncaught TypeError: foo is not a function(…)
}());
```

I dont get why the last console.log is a string?
And why when i invoke 'foo():' why is than foo not defined?

I hope some can explain this. Because im really stuck here.

Reply ↓

Anon *said* on 16/08/2017 at 2:58 am:

The value "foo" is pointing to (initially, a function definition) is updated during the code execution stage.

See my reply to Manx (below) for a more detailed explanation.

Reply ↓

Andi Wilson *said* on 09/09/2016 at 11:51 pm:

Hey, thank you for the write-up! Extremely helpful.

Just a quick typo in the last section:

"and when [we get] interpreter gets to var foo" should be

“and when [the] interpreter gets to var foo”

Reply ↓

Gibson Chikafa *said* on 21/09/2016 at 11:01 am:

Great article. I like it. Thank you!!

Reply ↓

MirSujat *said* on 26/09/2016 at 4:06 pm:

This is greate Article. Now i am clear about how js code work under the hood.. Thank you so much

Reply ↓

anson *said* on 20/10/2016 at 11:25 am:

I Just be here to say thankyou. Too wenderful.

Reply ↓

Manx *said* on 24/10/2016 at 9:39 pm:

var foo = 8;
function foo(){}
console.log(typeof foo); //number why?

Reply ↓

jeff *said* on 23/07/2017 at 6:27 pm:

it seems that the foo has been overwritten during the execution stage due to the same declared name, but it is not a good practice…

Reply ↓

Anon *said* on 16/08/2017 at 2:54 am:

var foo = 8; // A
function foo(){} // B
console.log(typeof foo); // C

During the creation stage a reference (property in the activation object/ variable object) named “foo” is created and points to the function declared in line B.

During the execution stage, the value “foo” points to is updated in line A. Now “foo” points to the number 8, NOT the function declared in line B.

Nothing happens in line B during the execution stage (nothing to be executed here).

In line C, “typeof” returns the type of the reference “foo” is pointing to, which, again, is the number 8.

Reply ↓

sameno *said* on 05/11/2016 at 10:49 am:

Hi David,

How does interpreter deal with alert statement in creation stage? Does it execute in create stage or execute stage?

Reply ↓

Anon *said* on 16/08/2017 at 2:43 am:

An “alert statement” is a function invocation like any other, of course it is going to be executed (hint) during the code execution stage.

Reply ↓

anson *said* on 22/11/2016 at 9:00 pm:

Hi, David. I have review this article many time.

I was confused this time with var vs let.

```
var style



function fn () {
  a = 1
  console.log(a)
  var a
}
fn() // The output was 1
```

```
let style
function fn () {
  a = 1
  console.log(a)
  let a
}
fn() // Will throw Error. a is not defined.
```

Does the scan of let is different from var?
thanks

Reply ↓

yan *said* on 08/06/2017 at 12:00 pm:

There is a temporal dead zone for let or const.

Reply ↓

Anon *said* on 16/08/2017 at 2:39 am:

http://exploringjs.com/es6/ch_variables.html#sec_temporal-dead-zone

“A variable declared by let or const has a so-called temporal dead zone (TDZ): When entering its scope, it can’t be accessed (got or set) until execution reaches the declaration.”

Reply ↓

Natasha *said* on 02/01/2017 at 1:14 pm:

I am sorry for my English.
Quote: Here is a pseudo-overview of how the interpreter evaluates the code:

1) Find some code to invoke a function.
2) Before executing the function code, create the execution context.
3) Enter the creation stage:
The end of the quote.
Because the creation stage is the stage of creation the execution context, the list item “Enter the creation stage” must depend of the list item number 2 – “Before executing the function code, create the execution context.” It cannot be independent of number 2.
Thanks.

Reply ↓

Natasha *said* on 04/01/2017 at 9:19 am:

```
fooExecutionContext = {
    scopeChain: { ... },
    variableObject: {
        arguments: {
            0: 22,
            length: 1
        },
        i: 22,
        c: pointer to function c()
        a: undefined,
        b: undefined
    },
    this: { ... }
}
fooExecutionContext = {
    scopeChain: { ... },
    variableObject: {
        arguments: {
            0: 22,
            length: 1
        },
        i: 22,
        c: pointer to function c()
        a: 'hello',
        b: pointer to function privateB()
    },
    this: { ... }
}
```

In my opinion the lines
c: pointer to function c()
and
b: pointer to function privateB()
look like little confused.
c pointes to function c (or variable c) because function c is declared and the variable c contents it's definition.
b pointes to function privateB(){}. The variable b contents FE therefore it's not correct to refer to its name (even linguistically) because it doesn't exist in global context. The expression function privateB() can be considered as a result of function execution. Because the variable b pointes to the FE, thr correct way to describe the subject of this pointing is the expression function privateB(){}.
Regards,
Natasha

Reply ↓

Omri *said* on 09/01/2017 at 9:58 pm:

Great Post!
Thanks

Reply ↓

CoderLIm *said* on 16/03/2017 at 2:56 pm:

Great Post, very thorough!

Reply ↓

Ray *said* on 02/05/2017 at 12:29 am:

I was watching Tony Alicia's excellent JS The Good Parts Udemy course and he really gets into the concept of the Execution Context, but I felt that there were some details he was glossing over (or I was missing). Then I found this article! All is clear now! Thank you.

Reply ↓

Levi *said* on 21/06/2017 at 12:25 am:

I want to know how the Execution Context deal with block scope, is it as same as function scope?

Reply ↓

Reaviel *said* on 23/06/2017 at 3:54 am:

Now i understand more about execution and creation context. Thank You.

Reply ↓

ufthelp *said* on 10/07/2017 at 10:05 pm:

Awesome Post. Thanks David Shariff for giving a crisp overview about execution context. Thanks.

Reply ↓

ufthelp *said* on 10/07/2017 at 10:06 pm:

Awesome Post . Thanks David for giving a crystal clear explanation.

Reply ↓

Pankaj *said* on 01/09/2017 at 12:07 am:

great article. very useful to me as noob. thanks

Reply ↓

Dedicated servers *said* on 11/09/2017 at 11:39 am:

What comes first, what comes second and so on, and if you did not have an “Execution Context” environment everything goes to hell.

Reply ↓

Coding System Design interviews *said* on 19/09/2017 at 2:46 am:

Awesome post!

Reply ↓

vivek kumar *said* on 24/09/2017 at 12:45 am:

```
var x = 10;

function z(){
var x = x = 20;
console.log(x);
}
z();
console.log(x); // why is 10 printed? Shouldn’t it be 20.
```

Reply ↓

Feng *said* on 28/09/2017 at 1:40 pm:

Wish I read this great aritical earlier.

BTW, is this IIFE (after “A Word On Hoisting”) a possible typo ?

```
}());
^
SyntaxError: Invalid or unexpected token
```

Reply ↓

Ravi Gupta ***said*** **on 29/09/2017 at 7:07 pm:**

Very through post. Great work

Reply ↓

duke63 ***said*** **on 12/10/2017 at 5:55 pm:**

very good explained, thanks

Reply ↓

Shahin ***said*** **on 06/11/2017 at 1:34 am:**

Thank you for this great article.

Could you please explain to me why we have ‘out’ log before ‘in’, here?

for (var i = 0; i < 5; i++) {
setTimeout(function () {
console.log('in '+ i);
}, 0);
}

console.log('out ' + i);

Reply ↓

Felix ***said*** **on 10/11/2017 at 4:08 pm:**

Awesome post! Cloud you tell me how you learn about this?reading book?

Reply ↓

Mahesh ***said*** **on 21/12/2017 at 6:39 pm:**

Hi,
Nice Article.
I have one doubt.
actually I am iterating an array in for loop.
I have nested if else in my for loop like this.
var arr={“one”: 1 ,”two” :2 ,”three” : 3};
for(key,value in arr)

```
{
if(key == “one”)
{
// some task
}
else if(“key” == “two”)
{
// some task
}
else
{
//some other task
}
}
```

what I want to know is
how the interpretation done and how the execution flow will be done?

actually in my case in the Interpretation process for keys one and two also it is coming to else block. but in execution process it is not coming to else block?
can you please explain in detail?
thank you in advance

Reply ↓

Spike *said* on 24/12/2017 at 12:31 pm:

thanks a lot

Reply ↓

Leave a Reply

Your email address will not be published. Required fields are marked *

Name *

Email *

Website

[Spam Check] What is: * + 3 = eight

Comment

Post Comment

← READ MORE