# Javascript Execution Context and Hoisting

**Is Javascript an Interpreted or a compiled language?**

Yes, Javascript (JS) is an **interpreted language**, still has its own form of a compiler, run in what’s known as the Javascript engine.

Every web browser has its own form of a JavaScript engine Eg. Chrome has v8 and Mozilla has spider monkey etc, though they all have the same purpose. JavaScript engines simply convert JavaScript source code into a language the compiler can understand, then executes it.

**Execution Context**

An environment in which the javascript code runs is what form an **execution context**.

The execution context decides what particular piece of code has access to variables, functions, objects, etc.

Execution context have different types —

1. **Global Execution Context (GEC)**
2. **Function Execution Context**

**1. Global Execution Context (GEC)**

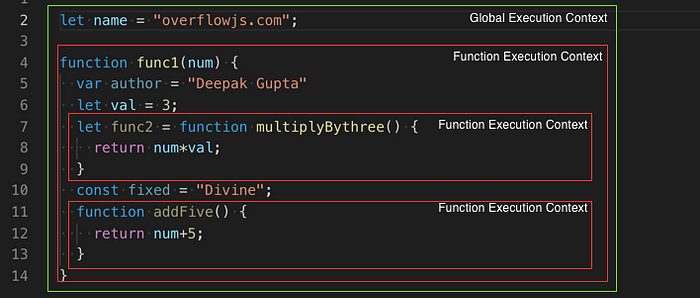
Whenever the code runs for the first time or when the code is not inside any function then it gets into the global execution context. There is only one global execution context throughout the execution of code.

In the case of browser global execution context does 2 things

1. Create a ‘window’ object.
2. The window object is referenced to ‘this’ keyword.

**2. Function Execution Context**

Whenever the code execution found a function it creates new function execution contexts. There can be any number of function execution contexts.



Above, Global execution context contains ‘name’ variable and a function reference to ‘func1’. Whereas three function execution context containing variables and function reference will be created. The Details is explained further in the article.

**Execution Stack / Call Stack**

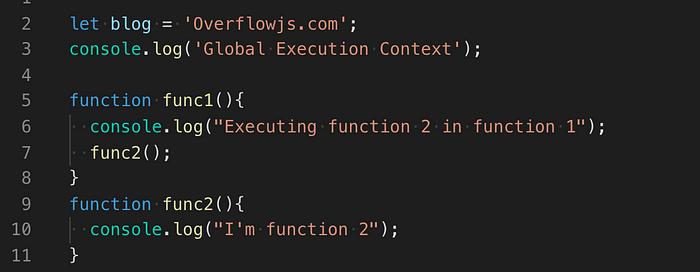
Javascript can only run one thing at one time in the browser that means it is the single thread so it queues the other action, events, function in what is called as the execution stack.

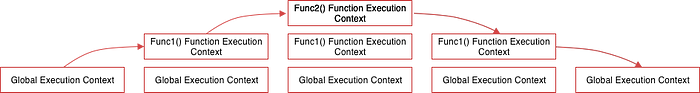
What is call stack in JavaScript engine?

The JavaScript engines use the call stack to keep track of what functions are running -- a function is added to a stack when the engine reaches a function that needs to be executed, and it cleared from the stack when the function returns.

Whenever a script load in the browser, the first element in the stack is the **global execution context**. However, when a function executes, an execution context is created and virtually placed on top of the global execution context. Once a function has finished executing, it gets popped off of the execution stack and returning control to the context below it.

Let’s take an example and visualize the above.





*Step1: When the above code loads in the browser, the Javascript engine creates a global execution context and pushes it to the current execution stack.*

*Step2: Let’s assume that at last, we do a func1() call then the Javascript engines creates a new execution context for that function and pushes it to the top of the global execution context*

*Step3: Inside the func1() we have func2() call therefore the Javascript engines creates a new execution context for that function and pushes it to the top of the func1 execution context.*

*Step4: When the func2() function finishes, its execution context is popped off from the current stack, and the control reaches the execution context below it, that is the func1() function execution context.*

*Step5: When the func1() finishes, its execution stack is removed from the stack and control reaches the global execution context. Once all the code is executed, the JavaScript engine removes the global execution context from the current stack.*

**Execution Context Phases**

There are mainly two phases of the execution context.

1. Creation
2. Execution

Let’s take a look one by one

**Creation phase**

There are several things that happen here before the function execution happen.

1. At first, a connection to the outer environment is created for each function or variables which is what form a scope chain. This tells the execution context what should it contain and where should it look for resolving the reference for function and values for variables.

* For the global environment, the outer environment is null. However, all environments within the global, have the global environment as its outer environment.
* If function ‘a’, contained in function ‘b’, that means ‘a’ has an outer environment ‘b’.

2. After scanning the scope chain an environment record is created where the creation and reference for global context(would be a window in a web browser), variable, function, and function arguments are done in memory.

3. At last value of ‘this’ keyword is determined (In the case of the global execution context, ‘this’ refers to the window) inside each execution context created in the 1st step.

**Execution Phase**

This is the phase where the code starts to run in the execution context formed in the creation phase and variable values are assigned line by line.

As the execution start, the engine looks for reference to execute the function in its creation phase object. If it doesn’t find it in its own, it will continue to move up the scope chain until it reaches the global environment.

If no references are found in the global environment it will return an error. However, if a reference is found and the function is executed correctly, the execution context of this particular function will be popped off the stack and the engine will move onto the next function, where their execution context will be added to the stack and executed, and so on.

Let's look at the above two phases via example to get a better idea around it.



So the global execution context will look something like this during the creation phase:

globalExecutionObj = {  
 outerEnvironmentConnection: null,  
 variableObjectMapping: {  
 name: uninitialized,  
 title: undefined,  
 date: uninitialized,  
 func1: func,  
 },  
 this: window //Global Object  
}

*Note: Above, the let (name)and const (date) defined variables do not have any value associated with them during the creation phase, but var (title)defined variables are set to undefined .*

This is the reason why you can access var defined variables before they are declared (though undefined) but get a reference error when accessing letand const variables before they are declared.

This is, what we call **hoisting** i.e all variable declarations using *var* are hoisted/lifted to the top of their functional/local scope (if declared inside a function) or to the top of their global scope (if declared outside of a function) regardless of where the actual declaration has been made.

During the execution phase, the variable assignments are done. So the global execution context will look something like this during the execution phase.

globalExectutionObj = {  
 outerEnvironmentConnection: null,  
 variableObjectMapping: {  
 name: "overflowjs.com",  
 title: "Execution context",  
 date: "5 july 2019",  
 func1: pointer to function func1,  
 },  
 this: window //Global Object  
}

Note: During the execution phase, if the JavaScript engine couldn’t find the value of let variable at the actual place it was declared in the source code, then it will assign it the value of undefined.

Now, when ‘func1’ is reached a new function execution context will be formed, whose creation object look like below

func1ExecutionObj = {  
 outerEnvironmentConnection: Global,  
 variableObjectMapping: {  
 arguments: {  
 0: 10,  
 length: 1  
 },  
 num: 10,  
  
 author: undefined,  
 val: uninitialized,  
 func2: undefined  
 fixed: uninitialized  
 addFive: pointer to function addFive()  
 },  
 this: Global Object or undefined  
}

During the execution phase,

func1ExecutionObj = {  
 outerEnvironmentConnection: Global,  
 variableObjectMapping: {  
 arguments: {  
 0: 10,  
 length: 1  
 },  
 num: 10,  
  
 author: "Deepak",  
 val: 3,  
 func2: pointer to function func2()   
 fixed: "Divine"  
 addFive: pointer to function addFive()  
 },  
 this: Global Object or undefined  
}

After the function completes its execution, the global environment is updated. Then the global code completes and the program finishes.

Use Javascript visualizer to see changes..