[Rupesh Mishra](https://hackernoon.com/@happymishra66?source=post_header_lockup)Follow

Full Stack Developer

May 19, 2017

**Execution context, Scope chain and JavaScript internals**

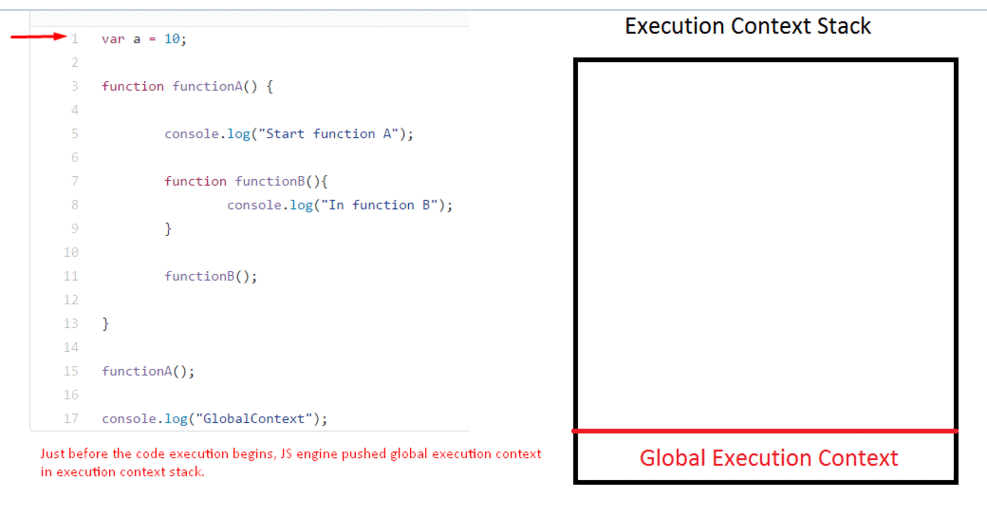
**Execution context (EC)**is defined as the environment in which JavaScript code is executed. By environment I mean the value of *this*, variables, objects, and functions JavaScript code has access to, constitutes it’s environment.

Execution context in JavaScript are of three types:

1. **Global execution context (GEC)**: This is the default execution context in which JS code start it’s execution when the file first loads in the browser. All the global code are executed inside global execution context. In the browser context, if the code is executing in strict mode value of *this*is *undefined*else it is *window*object. Global execution context cannot be more than one because only one global environment is possible for JS code execution.
2. **Functional execution context (FEC)**: Functional execution context is defined as the context created by the execution of code inside a function. Each function has it’s own execution context. It can be more than one. Functional execution context have access to all the code of global execution context. While executing global execution context code, if JS engine finds a function call, it creates a new functional execution context for that function.
3. **Eval**: Execution context inside eval function.

**Execution context stack (ECS):**Execution context stack is a stack data structure to store all the execution stacks created while executing the JS code. Global execution context is present by default in execution context stack and it is at the bottom of the stack. While executing global execution context code, if JS engines finds a function call, it creates functional execution context of that function and pushes that function execution context on top of execution context stack. JS engine executes the function whose execution context is at the top of the execution context stack. Once all the code of the function is executed, JS engines pop’s out that function’s execution context and start’s executing the function which is below it.

Let’s understand this with the help of an example:



As soon as the above code loads into the browser, JS engine pushes global execution context in the execution context stack. When functionA is called from global execution context, JS engines pushes functionA execution context in the execution context stack and starts executing functionA.

When functionB is called from functionA execution context, JS engine pushes functionB execution context in the execution context stack. Once all the code of functionB gets executed, JS engines pops out functionB execution context. As after this, functionA execution context is on top of the execution context stack, JS engine starts executing remaining code of functionA.

Once all the code from functionA gets executed, JS engines pops out functionA execution context from execution context stack and starts executing remaining code from the global execution context.

When all the code is executed JS engines pops out the global execution context and execution of JavaScript ends.

So far we have discussed how JavaScript engine handles the execution context. Now, we will see how JavaScript engine creates the execution context.

JavaScript engine creates the execution context in the following two stages:

1. Creation phase
2. Execution phase

Creation phase is the phase in which JS engines has called a function but it’s execution has not started. In the creation phase, JS engine is in the compilation phase and it scans over the function to compile the code.

In creation phase, JS engine performs the following task:

1. **Creates the Activation object or the variable object**: Activation object is a special object in JS which contain all the variables, function arguments and inner functions declarations information. As activation object is a special object it does not have the*dunder proto* property.
2. **Creates the scope chain:**Once the activation object gets created, JS engine initializes the scope chain which is a list of all the variables objects inside which the current function exists. This also includes the variable object of global execution context. Scope chain also contains the current function variable object.
3. **Determines the value of this:**After the scope chain, JavaScript engine initialize the value of *‘this’.*

Let’s understand how JavaScript engines creates the activation object with an example

Just after funA is called and before code execution of funA starts, JS engine creates an executonContextObj for funcA which can be represented as shown below:

Activation object or variable object contains argument object which have details about the arguments of the function.

It will have a property name for each of the variables and functions which are declared inside the current function

Activation object or the variable object in our case will be as shown below:

1. **ArgumentObject**: JS engines will create the argument object as shown in the above code. It will also have the length property indicating the total number of arguments in the function.
2. Now, for each variable in the function, JS engine will create a property on the Activation object and will initialize it with undefined. As arguments are also variables inside the function, they are also present as a property of the argument object.
3. If the variable already exists as a property of the argument object JS engine will not do anything and will move to the next line.
4. When JS engines encounters a function definition inside the current function, JS engine will create a new property by the name of the function. Function definitions are stored in heap memory, they are not stored in the execution context stack. Function name property points to it’s definition in the heap memory.

Hence, first **d**will get the value of *undefined*as it is a variable but when JS engine encounters a function with the same name it overrides it’s value to point it to the definition of function **d**stored in the heap.

After this JS engines will create the scope chain and will determine the value of *this*.

**Execution phase:**

In the execution phase, JS engines will again scan through the function to update the variable object with the values of the variables and will execute the code.

After the execution stage, variable object will look like this:

**Complete example:**

Consider the code below.

When the above code loads in the browser, JS engine will enter the compilation phase to create the execution objects. In the compilation phase JS engine will handle only the declarations, it does not bother about the values. This is the creation phase of execution context.

**Line 1**: In the line variable **a**is assigned a value of **1**, so JS engines does not think of it as a variable declaration or function declaration and it moves to line 3. It does not do anything with this line in compilation phase as it is not any declaration.

**Line 3**: As the above code is in global scope and it’s a variable declaration, JS engines will create a property with the name of this variable in the global execution context object and will initialize it with *undefined*value.

**Line 5**: JS engine finds a function declaration, so it will store the function definition in a heap memory and create a property which will point to location where function definition is stored. JS engines doesn’t know what is inside of **cFunc**.

**Line 13**: This code is not any declaration hence, JS engine will not do anything.

**Global Execution Context object after the creation phase stage:**

As further there is no code, JS engine will now enter the **execution phase** and will scan the function again. Here, it will update the variable value and will execute the code.

**Line 1**: JS engines find that there is no property with name **a**in the variable object, hence it adds this property in the global execution context and initializes it’s value to 1.

**Line 3**: JS engines checks that there is a property with name **b**in the variable object and hence update it’s value with 2.

**Line 5**: As it is a function declaration, it doesn’t do anything and moves to line 18.

**Global execution context object after the execution phase:**

**Line 18**: Here, **cFunc** is called, so JS engine again enters the compilation phase to create the execution context object of **cFunc**by scanning it.

As **cFunc**has **e**as an argument, JS engine will add **e**in the argument object of **cFunc**execution context object and create a property by the name of **e**and will initialize it with 2.

**Line 6**: JS engine will check if **c**is a property in the activation object of **cFunc**. As there is no property by that name, it will add **c**as a property and will initialize with *undefined*value.

**Line 7**: Same as line 6

**Line 9**: As this line is not a declaration, JS engine will move to next line

**Line 11**: JS engine finds a function declaration, so it will store the function definition in the heap memory and create a property **dFunc**which will point to location where function definition is stored. JS engines doesn’t know what is inside **dFunc**.

**cFunc execution context object after compilation phase:**

**Line 15**: As this statement is not a declaration, JS engine will not do anything.

As further there are no lines in this function JS engine will enter the execution phase and will execute **cFunc**by scanning it again.

**Line 6 and 7**: **c**and **d**get’s the value of 10 and 15 respectively

**Line 9**: As **a**is not a property on **cFunc**execution context object and it’s not a declaration, JS engine will move to the global execution context with the help of scope chain and checks if a property with name **a**exists in global execution context object. If the property does not exists, it will create a new one and will initialize it. Here, as property with **a**already exists on the global execution context object, it will updates it’s value to 3 from 1. JS engines moves to global execution context in this case only i.e. when it finds a variable in the execution phase which is not a property on the current execution context object

**Line 11**: JS engines will create a **dFunc**property and will points to it’s heap location

**Execution context object of cFunc after the execution phase:**

**Line 15**: As this is a function call, JS engines will again enter the compilation phase to create **dFunc**execution context object.

**dFunc**execution context object has access to all the variables and functions defined on **cFunc**and in the global scope using the scope chain.

Similarly **cFunc**has access to all the variables and objects in the global scope but it does not have any access to the **dFunc** variables and objects.

Global execution context does not have access to **cFunc**or **dFunc**variables or objects.

With the above concepts, I guess it’s will be easy to understand how **hoisting**works in JavaScript.

**Scope Chain:**

Scope chain is a list of all the variable objects of functions inside which the current function exists. Scope chain also consists of the current function execution object.

**Consider the below code:**

Here, when function **cFunc**is called from the global execution context, scope chain of **cFunc**will look like this

Scope chain of cFunc = [ cFunc variable object,   
 Global Execution Context variable object]

When **dFunc**is called from **cFunc**, as **dFunc**is inside **cFunc**, **dFunc’s**scope chain consists of **dFunc** variable object, **cFunc**variable object and **global execution context** variable object.

Scope chain of dFunc = [dFunc variable object,   
 cFunc variable object,  
 Global execution context variable object]

When we try to access **f**inside **dFunc**, JS engines checks if **f**is available inside **dFunc’s**variable object. If it finds **f’s**value it *console.log* **f’s**value.

When we try to access variable **c** inside **dFunc**, JS engines checks if **c**is available inside **dFunc’s** variable object. If the variable is not available, then it will move to **cFunc** variable object.

As variable **c** is not available inside **dFunc’s**variable object, JS engines moves to **cFunc’s**variable object. As **c**is available on **cFunc**variable object, it will *console.log* **c’s**value.

When we try to log **a’s**value inside **dFunc**, JS engines will check if **a**is available inside **dFunc’s** variable object. If **a**is not available inside **dFunc’s**variable object, it will move to the next item in scope chain i.e. **cFunc’s**variable object. JS engines will check if **cFunc’s** variable object as variable **a**. Here, variable **a** is not available on **cFunc’s**variable object hence, it will check the next items in **dFunc’s** scope chain i.e. **global execution context** variable object. Here **a** is available on **dFunc’s**variable object and it will console **a’ s**value.

Similarly, in case of **cFunc**, JS engine will find variable **a’s**value from global execution object.

**cFunc** does not know that variable **f** exists. Hence if we try to access **f**from **cFunc**it will give error. But, **dFunc**function has access **c** and **d**variable using the scope chain

Closures can be explained using the scope chain context in JavaScript.

**Other article:**

1. [Virtual DOM in ReactJS](https://medium.com/@happymishra66/virtual-dom-in-reactjs-43a3fdb1d130)
2. [Prototypes in JavaScript](https://medium.com/@happymishra66/prototypes-in-javascript-5bba2990e04b)
3. [‘this’ in JavaScript](https://hackernoon.com/this-in-javascript-8e8d4cd3930)
4. [Object.create in JavaScript](https://medium.com/@happymishra66/object-create-in-javascript-fa8674df6ed2)
5. [Inheritance in JavaScript](https://medium.com/@happymishra66/inheritance-in-javascript-21d2b82ffa6f)
6. [Create objects in JavaScript](https://medium.com/@happymishra66/create-objects-in-javascript-10924cfa9fc7)
7. [Objects in JavaScript](https://medium.com/@happymishra66/objects-in-javascript-2980a15e9e71)