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

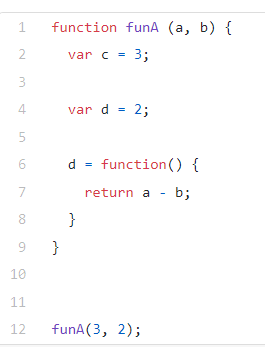
1. Creation phase
2. Execution phase

**Creation phase** is the phase in which the JS engine has called a function but its execution has not started. In the creation phase, JS engine is in the compilation phase and it just scans over the function code to compile the code, it doesn’t execute any code.

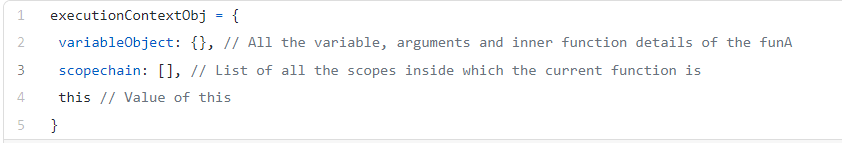
In the creation phase, JS engine performs the following task:

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 declaration 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, the 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 the global execution context. Scope chain also contains the current function variable object.
3. **Determines the value of this:** After the scope chain, the JavaScript engine initializes the value of this.

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

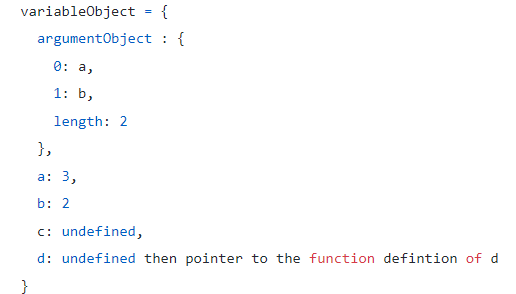


Just after funA is called and before code execution of funA starts, JS engine creates an executonContextObj for funA which can be represented as shown below:



Activation object or variable object contains the argument object which has 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. It will just have the property name, not its value
2. Now, for each variable in the function, JS engine will create a property on the activation object or variable 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 engine encounters a function definition inside the current function, it will create a new property by the name of the function. Function definitions in the creation phase are stored in heap memory, they are not stored in the execution context stack. Function name property points to its definition in the heap memory.

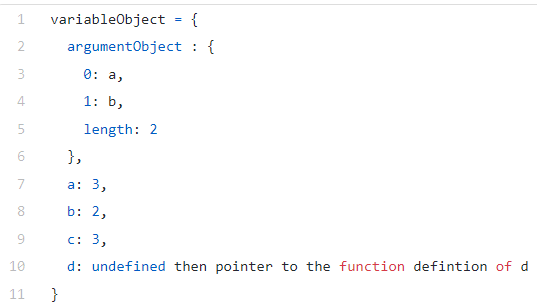
Hence in our case, first, **d**will get the value of undefined as it is a variable but when JS engine encounters a function with the same name it overrides its value to point it to the definition of function **d**stored in the heap.

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

**Execution phase:**

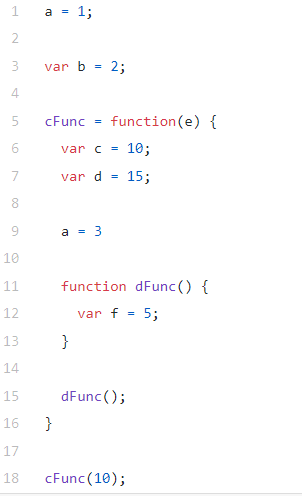
In the execution phase, JS engines will again scan through the function to update the variable object with the values of the variables and will execute the code.

After the execution stage, the 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 won’t bother about the values. This is the creation phase of the execution context.

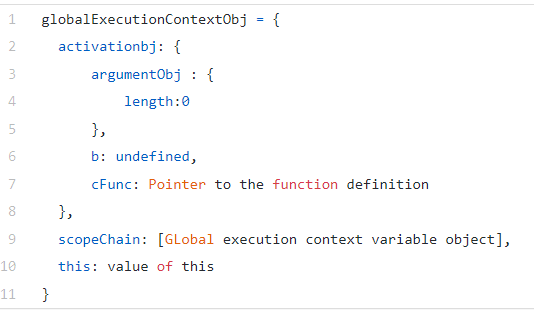
**Line 1**: In this line, variable**a**is assigned a value of **1**, so JS engine does not think of it as a variable declaration or function declaration and it moves to *line 3*. It does not do anything with this line in the compilation phase as it is not any declaration.

**Line 3**: As the above code is in the 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 an undefinedvalue.

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

**Line 18**: This code is not any declaration hence, JS engine will not do anything.

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



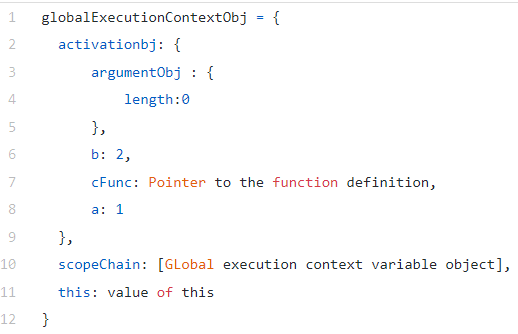
As further there is no code, JS engine will now enter the **execution phase** and will scan the function again. Here, it will update the variable value and will execute the code.

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

**Line 3**: JS engines checks that there is a property with the name**b**in the variable object and hence update its value to **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 cFuncby scanning it.

As cFunchas **e**as an argument, JS engine will add **e**in the argument object of cFuncexecution context object and create a property by the name of **e**and will initialize it to **2**.

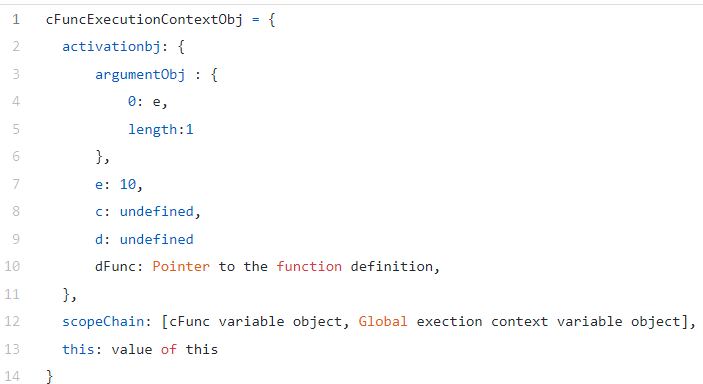
**Line 6**: JS engine will check if **c**is a property in the activation object of cFunc. As there is no property by that name, it will add **c**as property and will initialize its value to undefined.

**Line 7**: Same as line 6

**Line 9**: As this line is not a declaration, JS engine will move to the 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 dFuncwhich will point to the location where function definition is stored. JS engine doesn’t know what is inside dFunc.

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



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

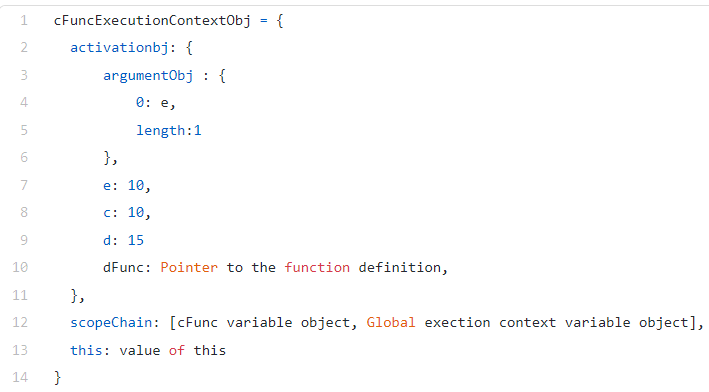
As further there are no lines in this function JS engine will enter the execution phase and will execute *cFunc*by scanning it again.

**Line 6 and 7**: **c**and **d**gets the value of 10 and 15 respectively

**Line 9**: As **a**is not a property on cFuncexecution 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 the name **a**exists in the global execution context object. If the property does not exist, it will create a new one and will initialize it. Here, as property with the name **a**already exists on the global execution context object, it will update its value to **3** from **1**. JS engine 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 dFuncproperty and will point to its 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 dFuncexecution context object.

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

Similarly, cFunchas 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 cFuncor dFuncvariables or objects.