**Q.2  Explain what is execution context in detail with diagram**

Ans. When the JavaScript engine scans a script file, it makes an environment called the **Execution Context**that handles the entire transformation and execution of the code.

During the context runtime, the parser parses the source code and allocates memory for the variables and functions. The source code is generated and gets executed.

There are two types of execution contexts: **global** and **function**. The global execution context is created when a JavaScript script first starts to run, and it represents the global scope in JavaScript. A function execution context is created whenever a function is called, representing the function's local scope.

**Phases of the JavaScript Execution Context**

There are two phases of JavaScript execution context:

1. **Creation phase**: In this phase, the JavaScript engine creates the execution context and sets up the script's environment. It determines the values of variables and functions and sets up the scope chain for the execution context.
2. **Execution phase**: In this phase, the JavaScript engine executes the code in the execution context. It processes any statements or expressions in the script and evaluates any function calls.

Everything in JS happens inside this execution context. It is divided into two components. One is memory and the other is code. It is important to remember that these phases and components are applicable to both global and functional execution contexts.

**Creation Phase**

Execution Context

Let's take this simple example once again:

var n = 5;

function square(n) {

var ans = n \* n;

return ans;

}

var square1 = square(n);

var square2 = square(8);

console.log(square1)

console.log(square2)

At the very beginning, the JavaScript engine executes the entire source code, creates a global execution context, and then does the following things:

1. Creates a global object that is**window** in the browser and **global** in NodeJs.
2. Sets up a memory for storing variables and functions.
3. Stores the variables with values as undefined and function references.

This is called the creation phase. Here's a diagram to help explain it:

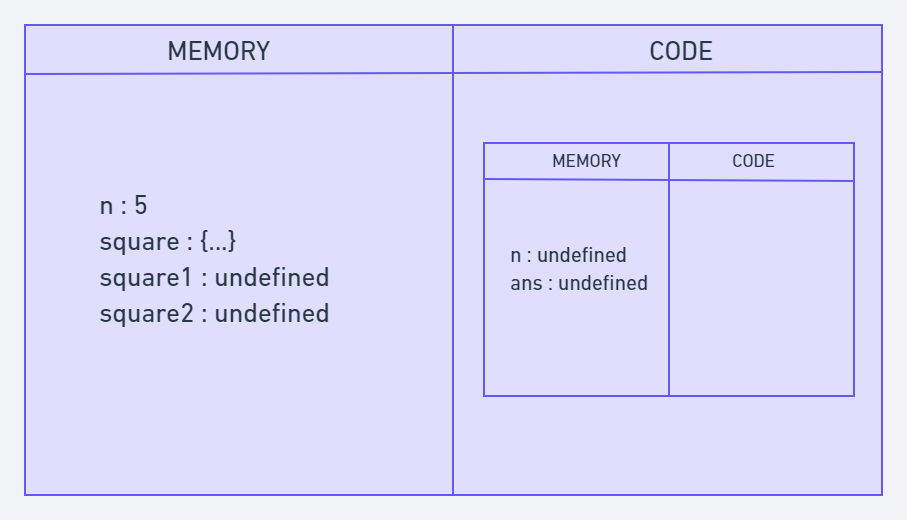
Creation Phase in Execution Context

After this creation phase, the execution context will move to the code execution phase.

**Execution Phase**

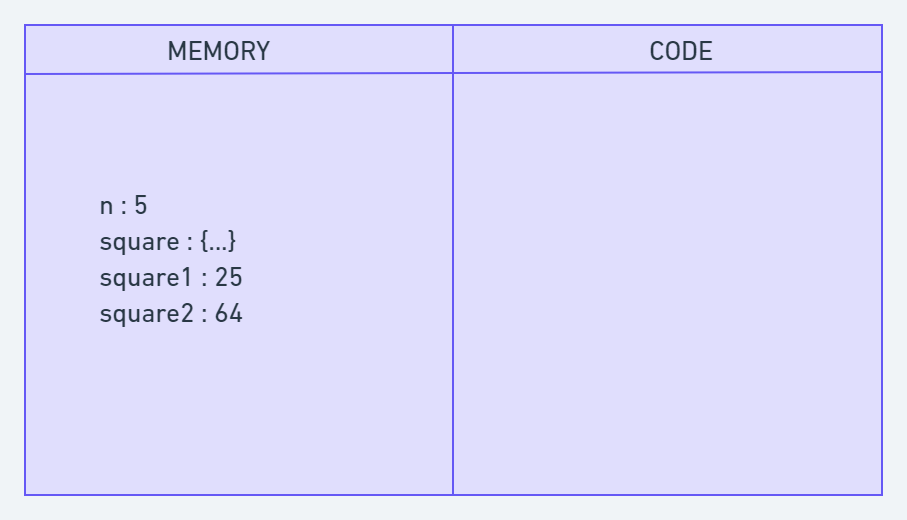
Now, in this phase, it starts going through the entire code line by line from top to bottom. As soon as it encounters **n = 5**, it assigns the value 5 to 'n' in memory. Until now, the value of 'n' was undefined by default.

Then we get to the 'square' function. As the function has been allocated in memory, it directly jumps into the line **var square1 = square(n);**. square() will be invoked and JavaScript once again will create a new function execution context.

Code Execution Phase

Once the calculation is done, it assigns the value of square in the 'ans' variable that was undefined before. The function will return the value, and the function execution context will be destroyed.

The returned value from square() will be assigned on square1. This happens for square2 also. Once the entire code execution is done completely, the global context will look like this and it will be destroyed also.



**What is the Call Stack?**

To keep the track of all the contexts, including global and functional, the JavaScript engine uses a **call stack**. A call stack is also known as an 'Execution Context Stack', 'Runtime Stack', or 'Machine Stack'.

It uses the LIFO principle (Last-In-First-Out). When the engine first starts executing the script, it creates a global context and pushes it on the stack. Whenever a function is invoked, similarly, the JS engine creates a function stack context for the function and pushes it to the top of the call stack and starts executing it.

When execution of the current function is complete, then the JavaScript engine will automatically remove the context from the call stack and it goes back to its parent.

Let's see the following example:

function funcA(m,n) {

return m \* n;

}

function funcB(m,n) {

return funcA(m,n);

}

function getResult(num1, num2) {

return funcB(num1, num2)

}

var res = getResult(5,6);

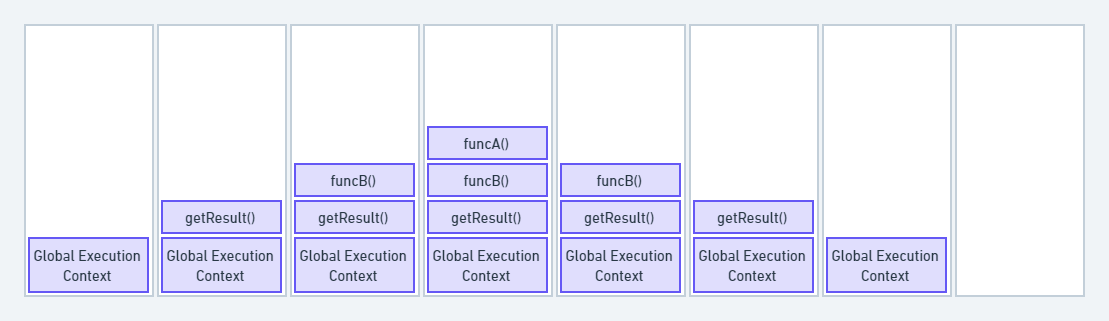
console.log(res); // 30

In this example, the JS engine creates a global execution context that enters the creation phase.

First it allocates memory for funcA, funcB, the getResult function, and the res variable. Then it invokes getResult(), which will be pushed on the call stack.

Then getResult() will call funcB(). At this point, funcB's context will be stored on the top of the stack. Then it will start executing and call another function funcA(). Similarly, funcA's context will be pushed.

Once execution of each function is done, it will be removed from the call stack. The following picture depicts the entire process of the execution:

Call Stack

The call stack has its own fixed size depending on the system or browser. If the number of contexts exceeds the limit, then a stack overflow error will occur. This happens with a recursive function that has no base condition.

function display() {

display();

}

display();

C:\Users\rwiteshbera\Desktop\Javascript\n.js:2

display();

^

RangeError: Maximum call stack size exceeded

Output

**Conclusion**

In conclusion, JavaScript execution context is a crucial part of understanding how JavaScript works behind the scenes. It determines the environment in which code is executed and what variables and functions are available to use.

The creation phase includes creating the global and function execution contexts, creating the scope chain, and allocating memories for the variables and functions. During the execution phase, the JavaScript engine executes the code line by line. This includes evaluating and executing statements.