

How JavaScript Code is Executed: A Deep Dive

When you run a JavaScript program, the JavaScript engine creates an **Execution Context** to manage the execution of the code. Let's break down how this works step by step with an example.

Example Code

```
Line1: var n = 2;
Line2: function square(num) {
Line3:   var ans = num * num;
Line4:   return ans;
Line5: }
Line6: var square2 = square(n);
Line7: var square4 = square(4);
```

Execution Context and Call Stack

JavaScript code is executed in two phases: 1. **Memory Allocation Phase** 2. **Code Execution Phase**

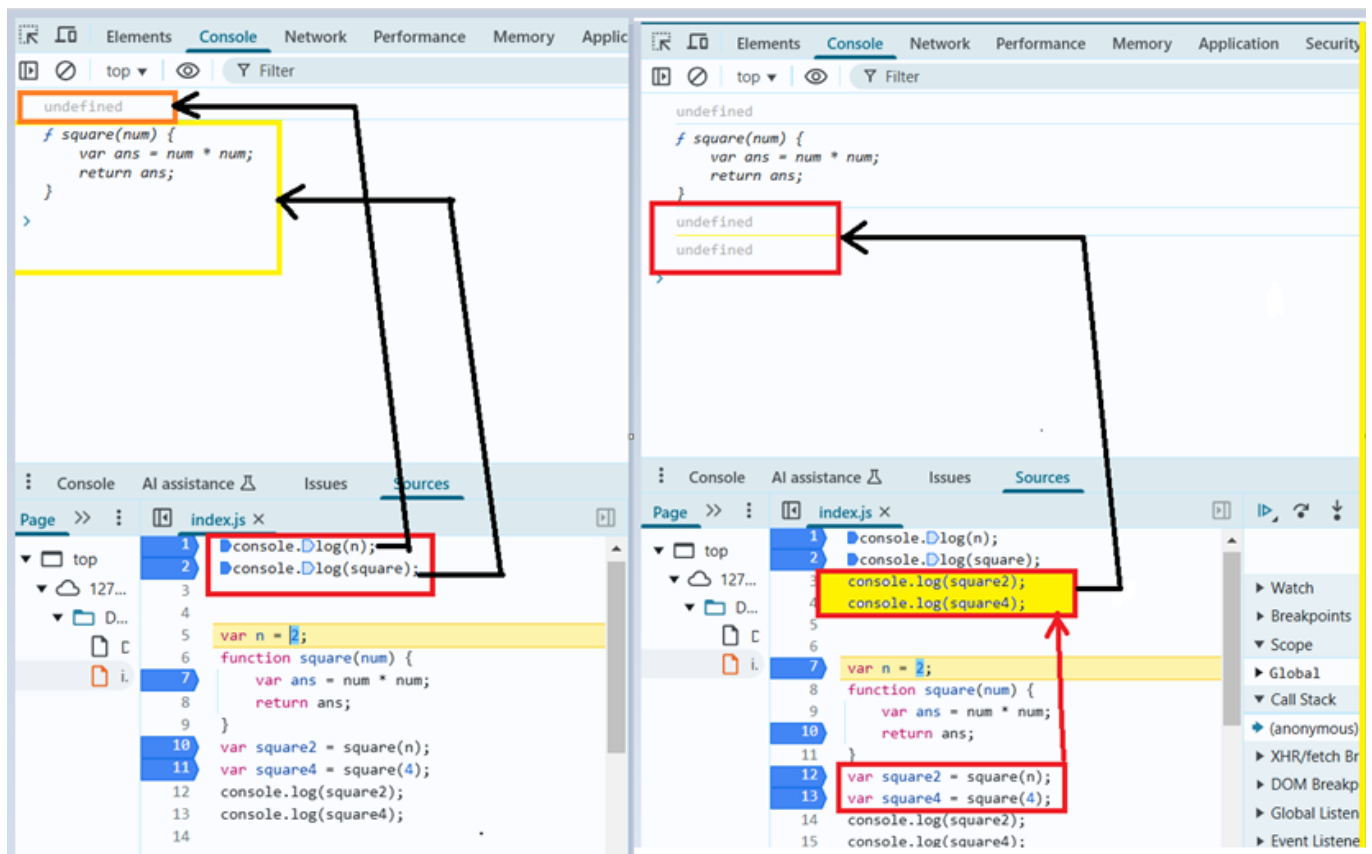
Each function invocation creates a new **Execution Context**, and the **Call Stack** manages the order of execution.

Step 1: Global Execution Context (GEC) - Memory Allocation Phase

Global Execution Context (GEC):

Memory	Code
n: undefined	
square: { ... }	(function)
square2: undefined	
square4: undefined	

- Variables are initialized with undefined.
- Functions are stored in memory as a whole.



Step 2: Global Execution Context (GEC) - Code Execution Phase

Line 1: `n = 2` is executed.

Global Execution Context (GEC):

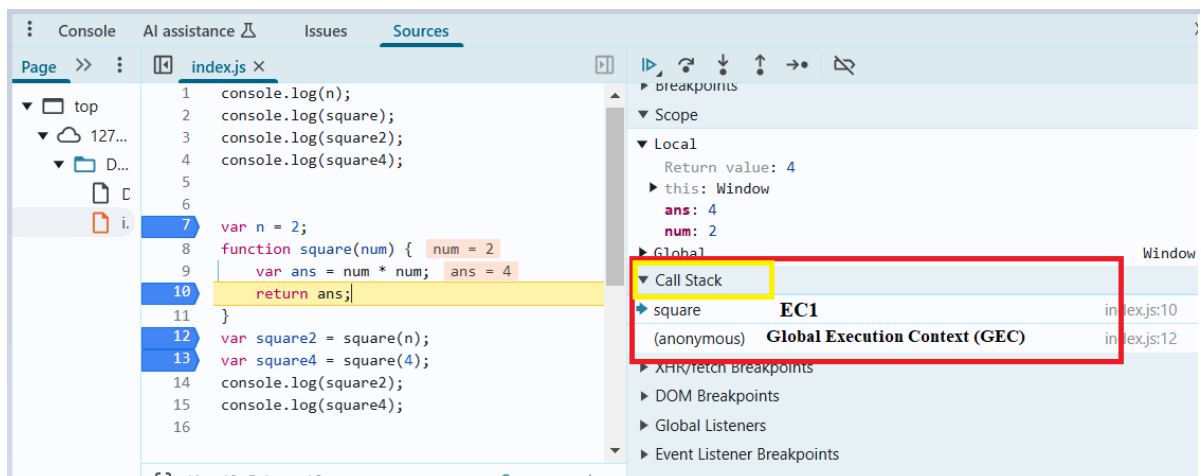
Memory	Code
<code>n: 2</code> <code>square: { ... }</code> <code>square2: undefined</code> <code>square4: undefined</code>	(function)

Line 6: `square(n)` is invoked.

- A new **Local Execution Context (EC1)** is created for `square`.

Call Stack:

```
-----  
| square EC1          |  
| Global EC (GEC)     |  
-----
```



Local Execution Context (EC1) - Memory Allocation Phase:

Local Execution Context (EC1):

```
-----  
| Memory          | Code          |  
-----  
| num: undefined  |              |  
| ans: undefined  |              |  
-----
```

Local Execution Context (EC1) - Code Execution Phase:

- `num` is assigned the value of `n` (which is 2).
- `ans = num * num` → `ans = 4`.
- `return ans` → The value 4 is returned to the GEC.

Local Execution Context (EC1):

Memory	Code

num: 2	
ans: 4	

- After execution, square2 is assigned the value 4 in the GEC.

Global Execution Context (GEC):

Memory	Code

n: 2	
square: { ... }	(function)
square2: 4	
square4: undefined	

- The Local Execution Context (EC1) is popped off the Call Stack.

Call Stack:

Global EC (GEC)	

Line 7: square(4) is invoked.

- A new **Local Execution Context (EC2)** is created for square.

Call Stack:

square EC2	
Global EC (GEC)	

Local Execution Context (EC2) - Memory Allocation Phase:

Local Execution Context (EC2):

Memory	Code
num: undefined	
ans: undefined	

Local Execution Context (EC2) - Code Execution Phase:

- num is assigned the value 4.
- $\text{ans} = \text{num} * \text{num} \rightarrow \text{ans} = 16$.
- `return ans` → The value 16 is returned to the GEC.

Local Execution Context (EC2):

Memory	Code
num: 4	
ans: 16	

- After execution, square4 is assigned the value 16 in the GEC.

Global Execution Context (GEC):

Memory	Code
n: 2	
square: { ... }	(function)
square2: 4	
square4: 16	

- The Local Execution Context (EC2) is popped off the Call Stack.

Call Stack:

```
-----
| Global EC (GEC) |
-----
```

Program Ends:

- The Global Execution Context (GEC) is popped off the Call Stack.

Call Stack:

```
-----
| (Empty) |
-----
```

Call Stack Visualization

The **Call Stack** maintains the order of execution contexts. Here's how it works:

1. Initial State:

Call Stack:

- Global Execution Context (GEC)

2. After square(n) is invoked:

Call Stack:

- square Execution Context (EC1)
- Global Execution Context (GEC)

3. After square(n) completes:

Call Stack:

- Global Execution Context (GEC)

4. After square(4) is invoked:

Call Stack:

- square Execution Context (EC2)
- Global Execution Context (GEC)

5. **After square(4) completes:**

Call Stack:

- Global Execution Context (GEC)

6. **Program Ends:**

Call Stack:

- (Empty)

Key Takeaways

- JavaScript execution is managed through **Execution Contexts** and the **Call Stack**.
- The **Memory Allocation Phase** sets up variables and functions.
- The **Code Execution Phase** runs the code and assigns values.
- Functions create their own Execution Contexts, which are managed in the Call Stack.