Global Execution Context

In JavaScript, the **Global Execution Context** is the default context in which all JavaScript code runs. It is the base execution environment that the JavaScript engine creates when a script is executed in a browser or a Node.js environment.

Here's a breakdown of what it involves:

1. Global Object Creation:

When the global execution context is created, a special object called the **global object** is also created. In the browser, this is the `window` object, and in Node.js, it is the `global` object. Any variable or function declared at the global level (i.e., outside of any function or block) becomes a property of this global object.

Example in a browser:

javascript
Copy code
```
var x = 10;
console.log(window.x); // 10
```

2. `this` Binding:

In the global execution context, the value of `this` refers to the **global object** (`window` in browsers).

Example:

javascript
Copy code
```
console.log(this === window); // true in the browser
```

3. Execution Phases:

The global execution context goes through two main phases:

- **Creation Phase:** During this phase, the JavaScript engine sets up the global object and the `this` keyword. It also sets aside memory for variables and function declarations (but not function expressions) and assigns them initial values (like `undefined` for variables).
- **Execution Phase:** In this phase, the JavaScript code starts executing line by line, and the previously declared variables and functions are assigned their actual values.

4. Single Global Execution Context:

There is only one global execution context in a JavaScript environment. It is created once when the script starts running and is destroyed when the script finishes.

Example of Global Execution Context:

javascript
Copy code

```javascript
var name = "Suyash";  // This is part of the global execution context

function greet() {
  console.log("Hello " + name);
}

greet();  // Runs in the global execution context
```

In this example, `name` and `greet` are both part of the global execution context because they are not defined inside any function or block. When `greet()` is called, it also executes within this context.

In summary, the global execution context is the environment in which all your top-level JavaScript code runs, and it governs the behaviour of variables, functions, and the `this` keyword in the global scope.

Function Execution Context

In JavaScript, a **Function Execution Context** is created whenever a function is invoked. This execution context is separate from the global execution context and defines the environment within which the function runs. Each function call generates its own execution context, which manages things like variable and function declarations within the function, and how the `this` keyword behaves inside the function.

Components of a Function Execution Context

When a function is called, its execution context is created in two phases: the **creation phase** and the **execution phase**. Here's how it works:

1. Creation Phase:

During the creation phase, the JavaScript engine sets up the following:

- **Argument Object:** This object contains the values passed to the function as arguments.
- **`this` Binding:** Determines what `this` refers to inside the function.
- **Memory for Local Variables and Functions:** The engine sets aside memory for variables and function declarations inside the function. These variables are initially set to `undefined` (due to hoisting), and functions are hoisted as well.

For example:

javascript
Copy code
```
function sayHello(name) {
  console.log(name);  // "Suyash"
}

sayHello("Suyash");
```

In this case, when the function `sayHello` is invoked, the argument `"Suyash"` is stored in the argument object, and the `name` variable is created and initialized with `"Suyash"`.

2. Execution Phase:

After the creation phase, the function enters the execution phase where the actual code within the function runs. During this phase, the variables are assigned the values, and the function code is executed line by line.

Key Components in Function Execution Context

Arguments Object: The function execution context creates an `arguments` object, which contains all the arguments passed to the function, even if the function doesn't explicitly list them in its parameters.
Example:

javascript
Copy code
```javascript
function sum() {
  console.log(arguments);
}

sum(1, 2, 3);  // Output: [1, 2, 3]
```

1.

Local Variables: Variables declared inside the function are stored in the function execution context, separate from the global scope.
javascript
Copy code
```javascript
function greet() {
  var greeting = "Hello";
  console.log(greeting);  // "Hello"
}

console.log(greeting);  // ReferenceError: greeting is not defined
```

2.

`this` Keyword: The value of `this` inside the function depends on how the function is called. If the function is called as a method of an object, `this` refers to that object; if the function is called in the global scope, `this` refers to the global object.
Example:
javascript
Copy code
```javascript
const person = {
  name: "Suyash",
  sayName: function() {
    console.log(this.name);  // "Suyash"
  }
};

person.sayName();  // "this" refers to the "person" object
```

3.
4. **Lexical Environment:** The function execution context also maintains a reference to its **Lexical Environment**, which includes its local environment (variables defined inside the

function) and a reference to the outer environment (the environment in which the function was defined, often the global scope or another function's execution context).

Execution Context Stack (Call Stack)

Each time a function is called, a new function execution context is created and pushed onto the **call stack**. When the function finishes execution, its execution context is popped off the stack. The JavaScript engine manages this stack to keep track of which context is currently executing.

Example of Function Execution Context in Action:

javascript
Copy code

```
var globalVar = "I am global";

function outerFunction(outerVar) {
  var outerVar2 = "Outer Variable 2";

  function innerFunction(innerVar) {
    console.log(globalVar);    // "I am global" (accesses global
variable)
    console.log(outerVar);     // "Outer Variable" (accesses outer
function variable)
    console.log(innerVar);     // "Inner Variable" (local to inner
function)
  }

  innerFunction("Inner Variable");
}

outerFunction("Outer Variable");
```

In this example:

- When `outerFunction` is invoked, a new execution context is created for it, which includes the variable `outerVar` and the definition of `innerFunction`.
- When `innerFunction` is called, another execution context is created for it, which has access to both its local variables and the outer function's variables.

Summary:

- A **Function Execution Context** is created every time a function is called.
- It goes through two phases: creation and execution.
- It contains the function's arguments, local variables, the value of `this`, and references to the outer environment.
- Execution contexts are managed by the **call stack**, where each function's context is pushed and popped as functions are called and returned.

What is an Exactly Lexical Environment ?

In JavaScript, a **Lexical Environment** refers to the structure that holds variable and function declarations within a particular scope, along with a reference to its **outer environment**. Essentially, it's where variables live and how the JavaScript engine keeps track of the scope in which variables are declared and how they should be accessed during execution.

Breakdown of Lexical Environment:

Local Environment: A **Lexical Environment** contains the local bindings of variables and functions defined within a specific block or function. These are stored as key-value pairs.
Example:
javascript
Copy code

```
function greet() {
  let name = "Suyash";  // Local variable
  console.log(name);    // "Suyash"
}

greet();   // The variable `name` exists in the lexical environment of
`greet`
console.log(name);  // ReferenceError: `name` is not defined
```

1. In the above example, `name` is only accessible within the lexical environment of the `greet` function. Outside the function, the variable is not accessible.

Reference to Outer Lexical Environment: Each Lexical Environment contains a reference to its **outer environment**, meaning the environment in which it was declared. This chain of environments allows for **lexical scoping**, where a function can access variables from its parent environment.
Example:
javascript

Copy code

```
let globalVar = "I am global";

function outerFunction() {
  let outerVar = "I am outer";

  function innerFunction() {
    let innerVar = "I am inner";
    console.log(globalVar);   // Can access outer environment
variable: "I am global"
    console.log(outerVar);    // Can access outer function variable:
"I am outer"
  }

  innerFunction();
}

outerFunction();
```

2. In this example:
   - `innerFunction` has its own lexical environment, which contains `innerVar`.
   - When `innerFunction` is called, it has access to its own `innerVar` and also the variables from `outerFunction` (`outerVar`) and the global environment (`globalVar`).

Lexical Scoping: Lexical scoping means that the scope of variables is determined by their physical location in the source code. In other words, the location where a function is written defines which variables it has access to, based on where it is nested.
Example:
javascript
Copy code

```
function outer() {
  let outerVar = "outer variable";

  function inner() {
    console.log(outerVar);  // Accesses outer variable due to lexical
scoping
  }

  inner();
}

outer();
```

3. In the above case, the `inner` function can access `outerVar` because of its lexical environment. Even though `inner` is executed later, it can access `outerVar` because it was defined within `outer`.

Two Main Parts of a Lexical Environment:

A Lexical Environment consists of two main parts:

1. **Environment Record:** This holds the actual bindings of variables and functions within the current environment. For instance, if you declare `let x = 10`, the environment record stores `x` and its value `10`.
2. **Outer Environment Reference:** This is a reference to the outer lexical environment, which allows the current environment to access variables and functions defined in outer scopes (parent environments).

Example of Lexical Environment Chain:

javascript
Copy code

```
let globalVar = "I am global";

function outerFunction() {
  let outerVar = "I am outer";

  function innerFunction() {
    let innerVar = "I am inner";
    console.log(innerVar);   // "I am inner" (Local environment)
    console.log(outerVar);   // "I am outer" (Outer environment
reference)
    console.log(globalVar);  // "I am global" (Global environment
reference)
  }

  innerFunction();
}

outerFunction();
```

Here’s the breakdown:

- The **global environment** contains `globalVar`.
- When `outerFunction` is called, a new **lexical environment** is created, which contains `outerVar`, and it has a reference to the global lexical environment.
- When `innerFunction` is called, another lexical environment is created for it, which contains `innerVar` and also has a reference to the lexical environment of `outerFunction`, allowing access to `outerVar` and `globalVar`.

Summary of Lexical Environment:

- A **Lexical Environment** is the structure that JavaScript uses to manage variable and function scopes.
- It contains local bindings (variables and functions) and a reference to the **outer environment**.
- Lexical scoping determines how variables are resolved: inner functions can access variables from outer functions, but outer functions cannot access variables from inner functions.
- This chain of environments ensures that JavaScript code can access variables from the correct scope, making the language behave predictably in terms of scope.

Eval Execution Context

In JavaScript, the `eval` function is used to evaluate a string of JavaScript code at runtime. When the `eval` function is called, it creates its own **Eval Execution Context** to process and execute the code inside the string.

What is the `eval` Execution Context?

The **Eval Execution Context** is created when you call `eval()` and pass a string of JavaScript code. This context is very similar to the **Global Execution Context** and **Function Execution Context**, but it's specifically for running code passed as a string inside `eval()`.

Here's how it works:

- The code inside the `eval()` is treated as if it were written in the same scope where the `eval()` is called.
- It creates its own execution context to process the code.
- Variables and functions declared inside `eval()` can affect the surrounding scope, which can make it harder to debug and maintain, which is why `eval()` is generally discouraged.

Example:

javascript
Copy code
```
let x = 5;

eval("x = 10;");  // Eval execution context created

console.log(x);  // Output: 10 (x is modified by eval)
```

In this case:

- `eval("x = 10;")` creates an eval execution context.
- The code inside the `eval()` changes the value of `x`, which affects the global variable `x`.

Key Features of Eval Execution Context:

1. **Dynamic Code Execution:** The `eval()` function can execute code passed to it as a string at runtime, making the code highly dynamic.
2. **Access to Surrounding Scope:** Code within `eval()` can access and modify variables in the same scope where `eval()` was called.
3. **Creation of Execution Context:** Just like functions, the `eval()` creates its own execution context with variable environment, lexical environment, and `this` binding.

How the Eval Execution Context Works:

When `eval()` is called:

1. **Creation Phase:**
   - The eval context is created, and the code inside the `eval()` string is prepared.
   - Variables and functions within the eval are hoisted just like in a regular function.
2. **Execution Phase:**
   - The code inside the eval string is executed.

Example with Local Scope:

javascript
Copy code
```
function testEval() {
  let a = 1;
  eval("let b = 2; console.log(a + b);");  // `a` is accessible inside
eval
}

testEval();  // Output: 3
```

In this example:

- The `eval()` creates a new execution context inside the function `testEval`.
- It can access `a`, which is outside the `eval()`, because `eval()` runs in the same scope as its surrounding code.

Why is Eval Execution Context Considered Risky?

- **Security Risks:** If you're using `eval()` on untrusted data (like user input), it can lead to code injection attacks.
- **Performance:** `eval()` slows down performance because the JavaScript engine cannot optimize the dynamically generated code in advance.
- **Debugging Difficulty:** Since the code is evaluated at runtime, it can make debugging harder and affect readability.

Summary:

- The `eval()` function creates its own **Eval Execution Context**, allowing the execution of JavaScript code contained in a string.
- It can modify the scope where it's called, making it powerful but also risky.
- It follows similar phases (creation and execution) as other execution contexts but is rarely used due to security and performance concerns.

Handwritten notes

Page No.:
Date:
YOUVA

How JavaScript execute code + Call Stack

↳ Histesh Sir

JavaScript Execution Context

$\boxed{\text{code } \{ \}} \rightarrow$ Global Execution Context

↳ this

so "this" is used. Global E.C is kept inside "this". To refer wherever Global Execution Context gets form, we use "this"

Global E.C of Browser & Nodejs is different. so in Browser, value of this is window object. JavaScript language is singe-threaded.

↳ Global Execution Content
↳ function Execution Content.
↳ Eval Execution Content
↳ This is property of Global Object

$\boxed{\{ \}}$ → Memory Creation Phase
→ Execution Phase

e.g.

```
let val1 = 10
let val2 = 5

fution addNum (num1, num2)
{
 let ~~[illegible]~~ total = num1 + num2;
   return total
}
 ~~let~~ let result1 = addNum (val1, val2)
     let result2 = addNum (4, 2)
```

Step 1: Global Execution / Environment

$$\begin{tikzcd} \text{Execution} \arrow[d] \\ \text{this binding} \end{tikzcd}$$

Step 2: Memory Creation Phase

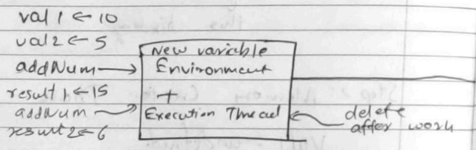
val1 = undefined.
val2 = undefined
addNum → definition
result1 → undefined
result 2 → undefined

Step 3: Execution Phase

val 1 ← 10
val2 ← 5
~~addNum~~ ← no execution task hai,
line 3 to line6 as you have already
hold the definition

~~result 1~~
line 7: result 1 ke andhar Jana hai
add no. lekin wo function hai
addNum alage Execution Content create
~~So addNum →~~ a
ek aur Execution content create
hogh

So,



Jitni baar function execute hongey,
utni baar Execution Contend.
So, again memory creation + execution
phase hongey

→ Memory Phase

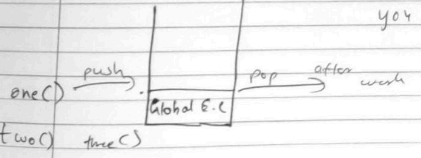
val1 → undefined
val 2 → undefined
total → undefined

Execution Content

~~val1~~ num1 → 10
num2 → 5
total → 15

total is returned
back to its
Parent Execution
Content.
↳ wese Global
Execution context
me return hota
hai, Sahi baat
hai Jab function
execute hoga &
wo return kar raha,
toh Global Execution
Content me return honge

Call Stack (LIFO)



If nested f()
, dont worry
you know concept
of recursion
wala.