

Day 1 JS Revision

This is my day 1 plan

Day 1 – Execution Context & Scope

Learn: Execution context, call stack, global scope, function scope, block scope.

Implement: Write examples with var, let, const.

Problems: Predict outputs of tricky scope/hoisting questions.

let me go through everything in details also I want to do practice on it

Perfect start 🚀. Day 1 is all about **Execution Context & Scope**, which is one of the most fundamental JS concepts. Let's break it into **theory** → **examples** → **practice problems** so you can revise deeply and build confidence.

♦ Step 1: Learn – Execution Context & Scope

1. Execution Context

When JS code runs, it creates **Execution Contexts**:

- **Global Execution Context (GEC):**
 - Created once when your file/script starts.
 - Creates the global object (`window` in browser, `global` in Node).
 - Binds `this` to global object (in non-strict mode).
 - Stores variables and functions declared globally.
- **Function Execution Context (FEC):**
 - Created each time a function is invoked.
 - Has its own **scope chain**, `this` **binding**, and **arguments object**.
 - Gets destroyed after function finishes (unless closures are involved).

📌 Each context has two phases:

1. Creation phase (Memory allocation / Hoisting):

- Variables with `var` are hoisted (set to `undefined`).
- Functions are hoisted completely.
- `let` & `const` are hoisted but in **temporal dead zone (TDZ)**.

2. Execution phase:

- Code runs line by line, assigning values and executing logic.

2. Call Stack

- JavaScript is **single-threaded** → executes one thing at a time.
- Uses a **call stack (LIFO)** to manage execution contexts.

Example:

```
js

function a() {
  console.log("inside a");
  b();
}
function b() {
  console.log("inside b");
}
a();
console.log("done");
```

Stack flow:

- `a()` pushed → executes → calls `b()` → pushes `b()` → pops after execution → finally pops `a()`.

3. Scope

- **Scope** = where variables are accessible.
 - **Global scope** – accessible everywhere.
 - **Function scope** – variables inside a function are private.
 - **Block scope** – created by `{ }` (with `let` & `const`).

4. var vs let vs const

- `var` → function-scoped, hoisted, allows redeclaration, initialized with `undefined`.

- `let` → block-scoped, hoisted but in TDZ, no redeclaration.
 - `const` → block-scoped, must be initialized immediately, cannot be reassigned.
-

◆ Step 2: Implement – Write Examples

js

// Global vs function scope

```
var x = 10;
function test() {
  var y = 20;
  console.log(x); // ✓ accessible
  console.log(y); // ✓ accessible
}
test();
console.log(y); // ✗ ReferenceError
```

// Block scope

```
if (true) {
  var a = 100;
  let b = 200;
  const c = 300;
}
console.log(a); // ✓ 100
console.log(b); // ✗ ReferenceError
console.log(c); // ✗ ReferenceError
```

// Hoisting with var

```
console.log(m); // undefined
var m = 5;
```

// Hoisting with let/const

```
console.log(n); // ✗ ReferenceError (TDZ)
let n = 10;
```

// Redeclaration

```
var p = 1;
var p = 2; // ✓ allowed
let q = 1;
let q = 2; // ✗ SyntaxError
```

◆ Step 3: Practice Problems

👉 Try predicting outputs **before** running them:

1.

```
js

console.log(a);
var a = 10;
function foo() {
  console.log(a);
  var a = 20;
}
foo();
console.log(a);
```

2.

```
js

{
  let x = 10;
  {
    console.log(x);
    let x = 20;
  }
}
```

3.

```
js

function test() {
  console.log(a);
  if (true) {
    var a = 5;
  }
  console.log(a);
}
test();
```

4.

```
js
```

```
for (var i = 0; i < 3; i++) {  
  setTimeout(() => console.log(i), 1000);  
}  
for (let j = 0; j < 3; j++) {  
  setTimeout(() => console.log(j), 1000);  
}
```

5.

js

```
function outer() {  
  let a = 10;  
  function inner() {  
    console.log(a);  
  }  
  return inner;  
}  
const fn = outer();  
fn(); // closure example
```

👉 My suggestion:

- Spend **1 hour studying notes + writing your own code examples.**
- Spend **1 hour on practice problems** (predict first, then run).
- Review mistakes & write down rules that you got wrong.

Do you want me to also **create 15–20 interview-style tricky scope/hoisting questions with solutions** so you can practice like a mock test after learning?

explain scope, scope chain and hoisting, lexical environment with more explanation and example

Awesome 🔥 this is exactly what makes the **foundation of JS** — *scope, scope chain, hoisting, and lexical environment*.

Let me break it down step by step with clear explanations + diagrams (conceptually) + code examples.

1. ♦ Scope

Definition:

Scope is the area where a variable is accessible. It defines the **lifetime and visibility** of variables.

Types of Scope:

1. Global Scope

Variables declared outside any function/block → accessible everywhere.

```
js

var globalVar = "I am global";

function show() {
  console.log(globalVar); // ✅ accessible
}

show();
console.log(globalVar); // ✅ accessible
```

2. Function Scope

Variables declared with `var` inside a function are local to that function.

```
js

function foo() {
  var x = 10;
  console.log(x); // ✅ 10
}

foo();
console.log(x); // ❌ ReferenceError
```

3. Block Scope (with `let` and `const`)

Variables declared with `let` or `const` inside `{ }` are only available inside that block.

```
js
```

```

if (true) {
  let y = 20;
  const z = 30;
  var w = 40;
}
console.log(w); // ✔ 40 (var ignores block scope)
console.log(y); // ✖ ReferenceError
console.log(z); // ✖ ReferenceError

```

2. ♦ Lexical Environment

Definition:

A **lexical environment** is the environment in which a piece of code is written.

It consists of:

- **Environment Record** → where variables/functions are stored.
- **Reference to the outer lexical environment** → which makes the **scope chain** possible.

📌 Each time you enter a new function/block, a new lexical environment is created.

Example:

```

js

function outer() {
  let a = 10;

  function inner() {
    let b = 20;
    console.log(a); // ✔ can access outer variable
    console.log(b); // ✔ can access own variable
  }

  inner();
}
outer();

```

Here:

- `inner()` has its own lexical environment (`b`).

- It also has a reference to `outer()`'s lexical environment (`a`).
 - This forms the **scope chain**.
-

3. ♦ Scope Chain

Definition:

When JavaScript tries to access a variable:

1. It first looks inside the **current lexical environment**.
2. If not found → goes to the **outer lexical environment**.
3. Continues until it reaches the **global scope**.
4. If still not found → `ReferenceError`.

Example:

```
js

let globalVar = "I am global";

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

  function inner() {
    let innerVar = "I am inner";
    console.log(innerVar); // ✅ found in inner
    console.log(outerVar); // ✅ found in outer
    console.log(globalVar); // ✅ found in global
  }

  inner();
}

outer();
```

Scope chain in action:

- `innerVar` → found inside `inner()`.
 - `outerVar` → not found in `inner`, so JS looks at `outer`.
 - `globalVar` → not found in `inner` or `outer`, so JS looks in global.
-

4. ♦ Hoisting

Definition:

Hoisting is JavaScript's default behavior of moving **declarations** to the top of their scope **during the creation phase** of execution context.

Rules:

- `var` is hoisted → initialized with `undefined`.
- `let` & `const` are hoisted → but in **Temporal Dead Zone (TDZ)** → can't be accessed before declaration.
- **Functions** are hoisted fully (can be used before they are declared).

Examples:

`var` Hoisting:

```
js

console.log(a); // undefined (not error!)
var a = 10;
```

This actually happens internally:

```
js

var a;    // hoisted
console.log(a); // undefined
a = 10;
```

`let` / `const` Hoisting (TDZ):

```
js

console.log(b); // ❌ ReferenceError
let b = 20;
```


Internally:

```
js

// b is hoisted but uninitialized (in TDZ)
let b;
console.log(b); // error
b = 20;
```

Function Hoisting:

```
js

greet(); //  works
function greet() {
  console.log("Hello");
}
```

But with function expressions:

```
js

sayHi(); //  TypeError: sayHi is not a function
var sayHi = function() {
  console.log("Hi");
};
```

5. ♦ Putting It All Together

👉 Example 1 – Mixed Hoisting & Scope:

```
js

console.log(x); // undefined
var x = 5;

function foo() {
  console.log(x); // undefined (local x is hoisted)
  var x = 10;
  console.log(x); // 10
}

foo();
console.log(x); // 5
```

👉 Example 2 – Scope Chain & Lexical Environment:

```
js

let a = "global";

function one() {
```

```

let b = "outer";

function two() {
  let c = "inner";
  console.log(c); // ✓ inner
  console.log(b); // ✓ outer
  console.log(a); // ✓ global
}

two();
one();

```

👉 Example 3 – TDZ Confusion:

```

js

function test() {
  console.log(a); // ✗ ReferenceError
  let a = 10;
}

test();

```

✓ So in short:

- **Scope** = visibility of variables.
- **Lexical Environment** = environment record + reference to parent.
- **Scope Chain** = link of lexical environments for variable lookup.
- **Hoisting** = declarations moved to top before execution.

Would you like me to now create a **big diagram-style explanation (ASCII / visual)** showing how lexical environments link together in a scope chain when nested functions are used? That will give you a crystal-clear mental model.

now just create the notes that we have learned so far and provide some question to practice,

Perfect 👍 let's consolidate everything into **crisp notes + practice questions** for Day 1 (Execution Context, Scope, Hoisting, Lexical Environment, Scope Chain).

Day 1 Notes – Execution Context & Scope

1. Execution Context

- The environment in which JS code runs.
- **Types:**
 - **Global Execution Context (GEC):** Created when the file runs. Contains global variables & functions.
 - **Function Execution Context (FEC):** Created when a function is invoked. Each has its own scope, `this`, and arguments.

Phases:

1. Creation phase:

- `var` → hoisted (initialized with `undefined`).
- `let` / `const` → hoisted but uninitialized (TDZ).
- Function declarations → hoisted fully.

2. Execution phase:

- Code runs line by line, assigning values and executing statements.

2. Call Stack

- JS is **single-threaded**.
- Uses a **call stack (LIFO)** to manage execution contexts.
- Function call → pushed on stack → executes → popped off after finishing.

3. Scope

- **Scope = visibility of variables.**
 - **Global Scope** → accessible everywhere.
 - **Function Scope (var)** → local to function.
 - **Block Scope (let, const)** → confined to `{ }`.

4. Lexical Environment

- **Lexical Environment** = variable environment + reference to parent environment.
 - Created whenever a function/block is invoked.
 - Each environment knows its parent → forms **scope chain**.
-

5. Scope Chain

- Variable lookup process:
 1. Check current scope.
 2. If not found → check parent scope.
 3. Continue until global scope.
 4. If still not found → `ReferenceError`.
-

6. Hoisting

- Process of moving **declarations** to top of scope during compilation.
 - **var** → hoisted & initialized with `undefined`.
 - **let/const** → hoisted but in TDZ.
 - **functions** → hoisted completely.
-

7. var vs let vs const

Feature	var	let	const
Scope	Function scope	Block scope	Block scope
Hoisting	Hoisted → <code>undefined</code>	Hoisted → TDZ	Hoisted → TDZ
Redeclaration	✓ Allowed	✗ Not allowed	✗ Not allowed
Reassignment	✓ Allowed	✓ Allowed	✗ Not allowed



Practice Questions

Q1. (Hoisting + Scope)

```
js

console.log(a);
var a = 10;
function foo() {
  console.log(a);
  var a = 20;
  console.log(a);
}
foo();
console.log(a);
```

Q2. (Block Scope + TDZ)

```
js

{
  console.log(x);
  let x = 10;
}
```

Q3. (Function Scope with var)

```
js

function test() {
  console.log(a);
  if (true) {
    var a = 5;
  }
  console.log(a);
}
test();
```

Q4. (Closures + Scope Chain)

js

```
function outer() {  
  let a = 10;  
  function inner() {  
    console.log(a);  
  }  
  return inner;  
}  
const fn = outer();  
fn();
```

Q5. (var vs let in loops)

js

```
for (var i = 0; i < 3; i++) {  
  setTimeout(() => console.log("var:", i), 1000);  
}  
  
for (let j = 0; j < 3; j++) {  
  setTimeout(() => console.log("let:", j), 1000);  
}
```

Q6. (Lexical Environment Chain)

js

```
let x = "global";  
  
function one() {  
  let y = "outer";  
  function two() {  
    let z = "inner";  
    console.log(z);  
    console.log(y);  
    console.log(x);  
  }  
  two();  
}
```

```
}  
one();
```

👉 **Task:** For each question, try to **predict the output before running it**.

That's how interviewers test your real understanding.

Would you like me to also create an **answer key with detailed explanations** for these questions so you can self-check after solving?

Explain this output

```
function outer() {  
  let a = 10;  
  function inner() {  
    console.log(a);  
  }  
  return inner;  
}  
const fn = outer();  
fn();
```

Great pick ⚡—this is one of the most fundamental **closure + lexical environment** examples. Let's go line by line and explain **why the output is 10**.

Code:

```
js  
  
function outer() {  
  let a = 10;  
  
  function inner() {  
    console.log(a);  
  }  
  
  return inner;
```



```
}
```

```
const fn = outer();  
fn();
```

Step-by-step execution:

1. When `outer` is declared

- Nothing runs yet, just a function definition stored in memory.

2. When `outer()` is called

- A **new execution context** for `outer` is created.
- Inside `outer`, a variable `a = 10` is declared.
- A function `inner` is declared (it forms a **closure** over `a`).
- `outer` then **returns the `inner` function, not executing it yet**.
- At this point, normally after a function ends, its variables (`a`) would be garbage collected.
BUT since `inner` is still referencing `a`, JS keeps that lexical environment alive.

👉 `const fn = outer();` → now `fn` holds a reference to the `inner` function.

3. When `fn()` is called

- This executes the `inner` function.
- `inner` needs to log `a`.
- It looks for `a` inside its own scope → not found.
- Then it goes up the **scope chain** to `outer`'s lexical environment.
- Finds `a = 10`. ✅

4. Output:

10

♦ Why didn't `a` disappear after `outer` finished?

Because of **closure**.

- A **closure** is formed when a function "remembers" the variables from its lexical scope even after the outer function has finished executing.
- Here, `inner` closed over `a`, keeping it alive in memory.

♦ Visual Model (simplified)

sql

Global Lexical Env

└── `outer()` function reference

`outer()` call creates:

Lexical Env of `outer`:

`a = 10`

`inner()` function reference

`inner()` has `[[Scope]]` → points to `outer`'s env

When `fn()` runs:

`inner()` looks for `a`

→ not found in `inner()`

→ found in `outer()` → prints `10`

✓ Key takeaways:

- **Lexical environment** is preserved due to closure.
- Even though `outer` finished, its variables remain alive as long as they're referenced by `inner`.
- Closures = functions + their surrounding state.

Do you want me to also show you a **modified version where `a` changes after returning**, so you can see how closures capture variables dynamically (not just their values at function creation time)?

yes

Perfect, let's see how closures behave when the variable changes **after** the function is returned. Closures don't just capture the value; they capture a **reference** to the variable in the lexical environment.

Example 1 – Variable Updated Before Call

```
js

function outer() {
  let a = 10;

  function inner() {
    console.log(a);
  }

  a = 20; // variable changed before returning

  return inner;
}

const fn = outer();
fn();
```

Execution:

- `outer()` runs → `a = 10` initially.
- Before returning, `a` is reassigned to `20`.
- `inner` closes over `a` (reference, not copy).
- When `fn()` executes → it sees the latest value of `a`.

Output:

```
20
```

Example 2 – Updating After Returning

```
js
```

```
function outer() {  
  let a = 10;  
  
  function inner() {  
    console.log(a);  
  }  
  
  return {  
    inner,  
    setA: (val) => { a = val; }  
  };  
}  
  
const obj = outer();  
obj.inner(); // prints 10  
obj.setA(42); // modifies `a` after outer has returned  
obj.inner(); // prints 42
```

Execution:

- First call → `inner` sees `a = 10`.
- Then `setA(42)` changes `a` even though `outer` has already finished.
- Closure keeps `a` alive → so next `inner()` call prints updated value.

Output:

```
10  
42
```

Example 3 – Multiple Closures Sharing Same Variable

```
js  
  
function counter() {  
  let count = 0;  
  
  return {  
    inc: () => { count++; },  
    dec: () => { count--; },  
  };  
}
```

```
    get: () => count
  };
}

const c = counter();
c.inc();
c.inc();
console.log(c.get()); // 2
c.dec();
console.log(c.get()); // 1
```

👉 Here all returned functions share the same `count` variable.

Closures make this possible → that's how **private variables** are implemented in JS.

✅ Key Takeaway:

Closures **don't copy values**; they **hold references** to variables in their lexical environment. If those variables change later, closures will always reflect the updated value.

Would you like me to also give you **some interview-style closure practice problems** (predict the output type), so you can test yourself on this?

```
now explain this
for (var i = 0; i < 3; i++) {
  setTimeout(() => console.log("var:", i), 1000);
}

for (let j = 0; j < 3; j++) {
  setTimeout(() => console.log("let:", j), 1000);
}
```

Ahh, this is a **classic interview trap** 🎯—and it directly connects to **scope + closures**. Let's break it step by step.

Code:

js

```
for (var i = 0; i < 3; i++) {  
  setTimeout(() => console.log("var:", i), 1000);  
}  
  
for (let j = 0; j < 3; j++) {  
  setTimeout(() => console.log("let:", j), 1000);  
}
```

◆ First Loop (with `var`)

- `var` is **function-scoped**, not block-scoped.
- That means the same single variable `i` is shared across all loop iterations.
- By the time the `setTimeout` callbacks actually run (after ~1000ms), the loop has already finished.
At that point:
 - `i = 3` (loop exited).
- All 3 callbacks reference the **same** `i` , which is now `3` .

👉 Output:

csharp

```
var: 3  
var: 3  
var: 3
```

◆ Second Loop (with `let`)

- `let` is **block-scoped**.
- Each iteration of the loop creates a **new lexical environment** with its own copy of `j` .
- So each `setTimeout` callback closes over a different `j` .
- After 1000ms, the callbacks log the values they captured at each iteration: `0` , `1` , `2` .

👉 Output:

```
bash
```

```
let: 0
```

```
let: 1
```

```
let: 2
```

◆ Combined Output (order may vary slightly, but conceptually):

```
csharp
```

```
var: 3
```

```
var: 3
```

```
var: 3
```

```
let: 0
```

```
let: 1
```

```
let: 2
```

Why does this happen?

- **var loop:** one shared variable `i`. All closures (the arrow functions inside `setTimeout`) reference that one variable. When loop finishes → `i = 3`.
- **let loop:** each iteration has its **own** `j`, so closures capture the value as it was during that iteration.

◆ Trick to Fix `var` Case Without `let`

If you wanted the `var` loop to behave like `let`, you can create a new scope in each iteration using an **IIFE** (Immediately Invoked Function Expression):

```
js
```

```
for (var i = 0; i < 3; i++) {  
  (function(x) {  
    setTimeout(() => console.log("fixed var:", x), 1000);  
  })(i);  
}
```

👉 Output:

```
csharp
```

```
fixed var: 0
```

```
fixed var: 1
```

```
fixed var: 2
```

Here, each iteration passes `i` into the IIFE, creating a new scope with its own copy

```
x.
```

✓ Key takeaway:

- `var` is function-scoped → shared variable across iterations.
 - `let` is block-scoped → new variable for each iteration.
 - Closures capture **references**, so with `var` they all reference the same variable, but with `let` they reference different variables.
-

Would you like me to also give you a **visual diagram (scope chain view)** of how `var` vs `let` differs in loops? That might lock this in permanently.