How JavaScript Works Behind the Scenes

> 🎓 **CS Perspective:** This section explores JavaScript's **execution model** and **runtime architecture**. The call stack implements a **LIFO (Last-In-First-Out)** data structure for managing execution contexts—directly related to how recursion and function calls work at the assembly level. Hoisting demonstrates the distinction between **compile-time** and **runtime** phases. The scope chain implements **lexical scoping** (static scoping), where variable resolution follows the source code structure. Understanding `this` binding reveals JavaScript's approach to **dynamic dispatch**—how method calls are resolved at runtime based on the calling context, not the definition location.

📌 JavaScript Engine & Runtime

JavaScript Engine

- **Heap**: Memory storage for objects
- **Call Stack**: Where code is executed (execution contexts)

JavaScript Runtime (Browser)

- JS Engine
- Web APIs (DOM, Timers, Fetch)
- Callback Queue
- Event Loop

📌 Execution Context

Created when code runs, contains:

1. **Variable Environment** - let, const, var, functions, arguments
2. **Scope Chain** - References to outer scopes
3. **this keyword** - Current context

Types

- **Global Execution Context** - Created for top-level code
- **Function Execution Context** - Created for each function call

---

📌 Scope & Scope Chain

Types of Scope

Scope	Access	Created By
Global	Everywhere	Top-level code
Function	Inside function	Function declaration

Scope	Access	Created By
Block	Inside `{}`	`if`, `for`, `while`

Scope Chain

```
const global = 'I am global';

function outer() {
  const outerVar = 'I am outer';

  function inner() {
    const innerVar = 'I am inner';
    console.log(global);    // ✅ Can access
    console.log(outerVar);  // ✅ Can access
    console.log(innerVar);  // ✅ Can access
  }

  inner();
  console.log(innerVar);    // ❌ ReferenceError
}
```

let/const vs var

```
if (true) {
  var varVariable = 'var';      // Function scoped
  let letVariable = 'let';      // Block scoped
  const constVariable = 'const'; // Block scoped
}

console.log(varVariable);    // ✅ 'var'
console.log(letVariable);    // ❌ ReferenceError
console.log(constVariable);  // ❌ ReferenceError
```

📌 Hoisting

Type	Hoisted	Initial Value	Scope
Function Declaration	✅ Yes	Actual function	Block
var	✅ Yes	`undefined`	Function
let/const	✅ Yes	TDZ (uninitialized)	Block
Function Expression	Depends on var/let/const		
Arrow Function	Depends on var/let/const		

Temporal Dead Zone (TDZ)

```
console.log(x); // ❌ ReferenceError (TDZ)
let x = 10;
```

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

Function Hoisting

```
// ✅ Works - function declarations are hoisted
greet();
function greet() {
  console.log('Hello!');
}

// ❌ Error - expressions not hoisted
greet2(); // TypeError
const greet2 = function() {
  console.log('Hello!');
};
```

📌 The `this` Keyword

`this` depends on **how** the function is called, not where it's defined.

Rules

Context	`this` Value
Global scope	`window` (browser)
Regular function	`undefined` (strict mode)
Arrow function	`this` of parent scope
Method	Object calling the method
Event listener	DOM element
`call`/`apply`/`bind`	Specified object

Examples

```
'use strict';

// Global
console.log(this); // window

// Regular function
const calcAge = function(birthYear) {
  console.log(this); // undefined (strict mode)
};

// Arrow function (inherits from parent)
const calcAgeArrow = birthYear => {
  console.log(this); // window (from global)
};

// Method
const jonas = {
  year: 1991,
  calcAge: function() {
    console.log(this);      // jonas object
    console.log(this.year); // 1991
  }
};
jonas.calcAge();
```

📌 Regular Functions vs Arrow Functions

`this` Behavior

```
const jonas = {
  firstName: 'Jonas',
  year: 1991,

  // Method with regular function
  calcAge: function() {
    console.log(this); // jonas object ✅

    // Problem: nested function loses this
    const isMillenial = function() {
      console.log(this); // undefined ❌
    };

    // Solution 1: self = this
    const self = this;
    const isMillenial2 = function() {
      console.log(self.year); // ✅ Works
    };

    // Solution 2: Arrow function (inherits this)
```

```
    const isMillenial3 = () => {
      console.log(this.year); // ✅ Works
    };
  },

  // ⚠ Don't use arrow for methods!
  greet: () => {
    console.log(this); // window, not jonas! ❌
  }
};
```

Arguments Keyword

```
// Regular function has arguments
const addExpr = function(a, b) {
  console.log(arguments); // [2, 5, 8, 12]
  return a + b;
};
addExpr(2, 5, 8, 12);

// Arrow function does NOT have arguments
const addArrow = (a, b) => {
  console.log(arguments); // ❌ ReferenceError
  return a + b;
};
```

📌 Primitives vs Objects (Reference Types)

Primitives (Stored in Call Stack)

- Number, String, Boolean, Undefined, Null, Symbol, BigInt

Objects/Reference Types (Stored in Heap)

- Object literals, Arrays, Functions, etc.

Behavior Difference

```
// Primitives - Copy value
let age = 30;
let oldAge = age;
age = 31;
console.log(age);    // 31
console.log(oldAge); // 30 (unchanged)

// Objects - Copy reference
const me = { name: 'Jonas' };
const friend = me;
```

```
friend.name = 'Steven';
console.log(me.name);     // 'Steven' (changed!)
console.log(friend.name); // 'Steven'
```

📌 Copying Objects

Shallow Copy

```
const jessica = {
  firstName: 'Jessica',
  family: ['Alice', 'Bob']
};

// Spread operator (shallow)
const jessicaCopy = { ...jessica };
jessicaCopy.firstName = 'Jess'; // ✅ Won't affect original

jessicaCopy.family.push('Mary'); // ❌ WILL affect original!
console.log(jessica.family); // ['Alice', 'Bob', 'Mary']
```

Deep Clone

```
// Modern solution: structuredClone
const jessicaClone = structuredClone(jessica);
jessicaClone.family.push('John');

console.log(jessica.family);      // ['Alice', 'Bob'] ✅
console.log(jessicaClone.family); // ['Alice', 'Bob', 'John'] ✅
```

🧪 Quick Reference Cheatsheet

```
// Scope
// var = function scoped
// let/const = block scoped

// Hoisting
// Functions: fully hoisted
// var: hoisted as undefined
// let/const: TDZ (temporal dead zone)

// this keyword
// Method: object calling it
// Regular function: undefined (strict)
// Arrow function: inherited from parent
```

```
// Event listener: DOM element

// Copying
{ ...obj }              // Shallow copy
structuredClone(obj)    // Deep clone

// Check property ownership
obj.hasOwnProperty('key');
```

✅ Exam Tips

1. **let/const are block scoped**, var is function scoped
2. **TDZ** = Temporal Dead Zone (let/const before declaration)
3. **Function declarations** are hoisted, expressions are not
4. **`this` in arrow functions** = inherited from surrounding scope
5. **Never use arrow functions as methods** - `this` will be wrong
6. **Objects are reference types** - copying creates a reference, not a new object
7. **`structuredClone()`** for deep cloning (modern JS)
8. **Spread operator** `{...obj}` creates shallow copy only
9. Arrow functions don't have **`arguments`** object
10. **Strict mode** makes `this` = `undefined` in regular functions