JavaScript Day 5 - Complete Notes

📋 Table of Contents

🔄 Type Coercion

Type coercion is the **automatic or explicit conversion of one data type to another** in JavaScript. This is a fundamental concept that affects how JavaScript handles operations between different data types.

Types of Type Coercion

1. Implicit Coercion (Automatic)

JavaScript engine automatically converts data types when needed.

```
const a = "10";    // String
const b = 5;       // Number

const c = a * b;   // Result: 50 (Number)
// JavaScript automatically converts string "10" to number 10
console.log("c = " + c);        // Output: "c = 50"
console.log(typeof c);          // Output: "number"
```

How it works:

- JavaScript sees multiplication operator (`*`)
- Realizes it needs numbers to perform multiplication
- Converts string "10" to number 10
- Performs 10 * 5 = 50

2. Explicit Coercion (Manual/Forced)

Programmer deliberately converts data types using built-in functions.

```
const stringNumber = "25";

// Explicit conversions
const toNumber = Number(stringNumber);      // 25 (number)
const toString = String(123);               // "123" (string)
const toBoolean = Boolean(1);               // true (boolean)
const toBigInt = BigInt(42);                // 42n (bigint)
```

Bonus Conversion Techniques

The Unary Plus Operator (`+`)

A shortcut for converting to numbers:

```
const a = "10";
const z = +a;           // Equivalent to Number(a)
console.log(typeof z);  // Output: "number"
console.log(z);         // Output: 10
```

Prompt with Automatic Conversion

```
// Without conversion
let data = prompt("Enter User Age: ");
console.log(typeof data);  // "string"

// With automatic conversion
let age = +prompt("Enter User Age: ");
console.log(typeof age);   // "number"
```

The typeof Operator

The `typeof` operator returns a string indicating the type of a variable or expression.

Syntax

```
typeof variable
typeof(variable)  // Alternative syntax with parentheses
```

Common Return Values

```
console.log(typeof 42);           // "number"
console.log(typeof "hello");      // "string"
console.log(typeof true);         // "boolean"
console.log(typeof undefined);    // "undefined"
console.log(typeof null);         // "object" (this is a known quirk!)
console.log(typeof {});           // "object"
console.log(typeof []);           // "object"
console.log(typeof function(){}); // "function"
```

🌐 Scope in JavaScript

Scope determines **where variables can be accessed** in your code. Think of scope as the "visibility" or "accessibility" of variables.

1. Global Scope

Variables declared outside any function or block have global scope.

```
var globalVar = "I'm global!";
let globalLet = "I'm also global!";
const globalConst = "Me too!";

function anyFunction() {
    console.log(globalVar);    // ☑ Accessible
    console.log(globalLet);    // ☑ Accessible
    console.log(globalConst);  // ☑ Accessible
}
```

2. Local Scope (Block Scope)

Variables declared inside a block `{}` have local scope.

```
{
    let blockScoped = "I'm local!";
    const alsoBlockScoped = "Me too!";
    var notBlockScoped = "I'm not truly local!";
}

console.log(blockScoped);      // ❌ Error: not defined
console.log(alsoBlockScoped);  // ❌ Error: not defined
console.log(notBlockScoped);   // ☑ Accessible (var ignores block scope)
```

3. Function Scope

Variables declared inside a function are only accessible within that function.

```
function myFunction() {
    var functionScoped = "I'm function scoped!";
    let alsoFunctionScoped = "Me too!";
    const meToo = "And me!";
}

console.log(functionScoped);  // ❌ Error: not defined
```

4. Script Scope

Variables declared with `let` and `const` at the top level create script scope.

📊 Scope Hierarchy Visualization

```
🌍 Global Scope
├── var globalVar = 10
├── let globalLet = 20
└── const globalConst = 30
    │
    └── 📦 Local Block Scope
        ├── var blockVar = 40      (actually goes to global!)
        ├── let blockLet = 50      (truly block scoped)
        └── const blockConst = 60  (truly block scoped)
            │
            └── 🔍 Inner Block Scope
                ├── let innerLet = 70
                └── const innerConst = 80
```

🔍 Variable Lookup Process

When JavaScript looks for a variable, it follows this process:

1. **Current Scope**: Look in the current block/function
2. **Parent Scope**: If not found, look in the parent scope

3. **Global Scope**: If still not found, look in global scope
4. **Error**: If nowhere, throw `ReferenceError`

Example of Scope Chain

```
var a = 1;         // Global scope
let b = 2;         // Script scope
const c = 3;       // Script scope

{
    // Local block scope
    console.log("Inside block:");
    var a = 10;    // Updates global 'a'
    let b = 20;    // Creates new block-scoped 'b'
    const c = 30; // Creates new block-scoped 'c'

    console.log("a:", a);  // 10 (local)
    console.log("b:", b);  // 20 (local)
    console.log("c:", c);  // 30 (local)
}

console.log("Outside block:");
console.log("a:", a);  // 10 (global was updated!)
console.log("b:", b);  // 2 (script scope unchanged)
console.log("c:", c);  // 3 (script scope unchanged)
```

💡 Key Differences: var vs let vs const

Feature	var	let	const
Scope	Function/Global	Block	Block
Hoisting	Yes (undefined)	Yes (TDZ)	Yes (TDZ)
Redeclaration	☑ Allowed	❌ Not allowed	❌ Not allowed
Reassignment	☑ Allowed	☑ Allowed	❌ Not allowed
Block Scope	❌ No	☑ Yes	☑ Yes

🗑️ Memory Management

Garbage Collection Process

When JavaScript engine control exits from any scope:

1. **Scope Destruction**: The scope is marked for destruction
2. **Variable Cleanup**: All variables in that scope become unreachable
3. **Garbage Collection**: Memory is automatically freed
4. **Memory Optimization**: Space is made available for new variables

```
function createScope() {
    let largeData = new Array(1000000);  // Takes memory
    const temporaryVar = "temporary";

    // When function ends, largeData and temporaryVar
    // are automatically cleaned up by garbage collector
}

createScope();  // After execution, memory is freed
```

🚀 Best Practices

1. Variable Declaration

```
// ☑ Good: Use const by default
const userName = "John";

// ☑ Good: Use let when you need to reassign
let counter = 0;
counter++;

// ❌ Avoid: var has confusing scoping rules
var oldStyle = "avoid this";
```

2. Scope Management

```
// ☑ Good: Keep variables in smallest possible scope
{
    const temporaryCalculation = x * y;
    return temporaryCalculation + z;
}

// ❌ Bad: Unnecessary global variables
var globalTemp = x * y;  // Pollutes global scope
```

3. Type Coercion

```
// ☑ Good: Explicit conversion
const userAge = Number(prompt("Enter age:"));

// ❌ Risky: Relying on implicit conversion
const userAge = prompt("Enter age:") * 1;
```

📝 Practice Examples

Example 1: Scope Chain

```
const global = "I'm global";
function outer() {
    const outerVar = "I'm outer";
    function inner() {
        const innerVar = "I'm inner";
        console.log(global);    // ☑ Accessible
        console.log(outerVar);  // ☑ Accessible
        console.log(innerVar);  // ☑ Accessible
    }
    inner();
    console.log(innerVar);  // ❌ Error: not accessible
}
outer();
```

Example 2: Type Coercion in Practice

```
// Implicit coercion examples
console.log("5" + 3);      // "53" (string concatenation)
console.log("5" - 3);      // 2 (numeric subtraction)
console.log("5" * 3);      // 15 (numeric multiplication)
console.log(true + 1);     // 2 (boolean to number)
console.log(false + 1);    // 1 (boolean to number)

// Explicit coercion examples
console.log(Number("5") + 3);    // 8
console.log(String(5) + 3);      // "53"
console.log(Boolean(1));         // true
console.log(Boolean(0));         // false
```

🎯 Summary

- **Type Coercion** allows JavaScript to convert between data types automatically (implicit) or manually (explicit)
- **typeof** operator helps identify the current type of a value
- **Scope** determines where variables can be accessed in your code
- **Block scope** (let/const) is more predictable than function scope (var)
- **Garbage collection** automatically manages memory when variables go out of scope
- Always prefer explicit type conversion and proper scope management for cleaner, more predictable code

---

Remember: Understanding scope and type coercion is crucial for writing robust JavaScript applications and avoiding common pitfalls!