Let’s Learn JS Fundamentals

Testleaf
Always Ahead

Agenda – Part 1 (Basics)

- Learn variables & Data types
- Primitive vs Non primitive data types
- Null vs Undefined
- var vs let vs const
- Re-declaration and Re-assignment
- Scoping
- Hoisting

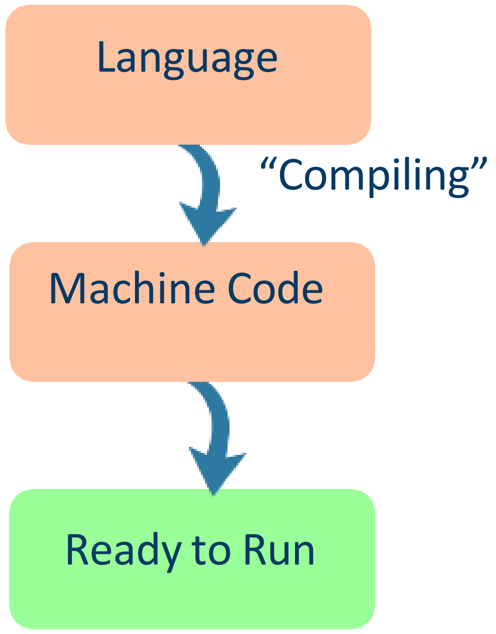
Compiled vs Interpreted language

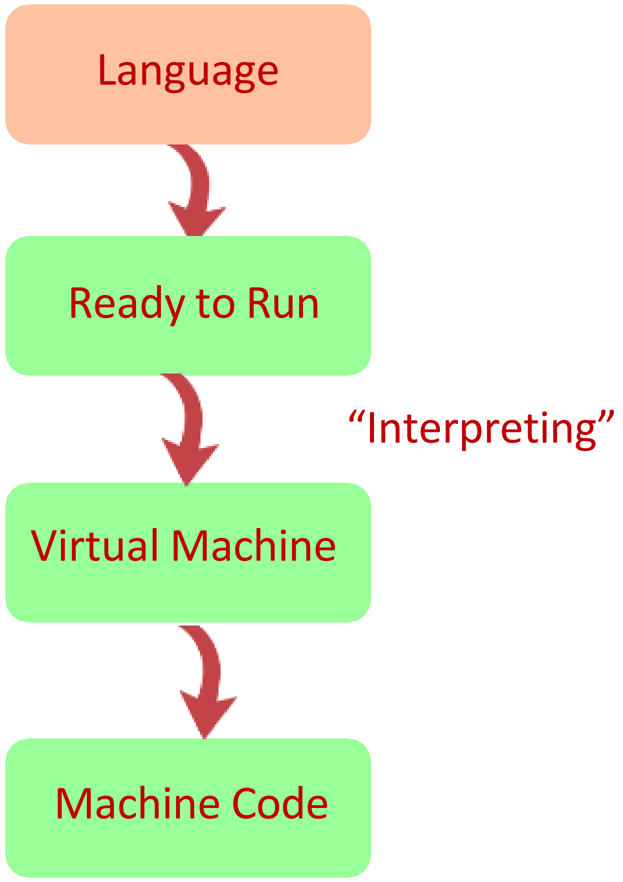
Compiled Language

C, C++, Fortran, Pascal, Java

Interpreted Language

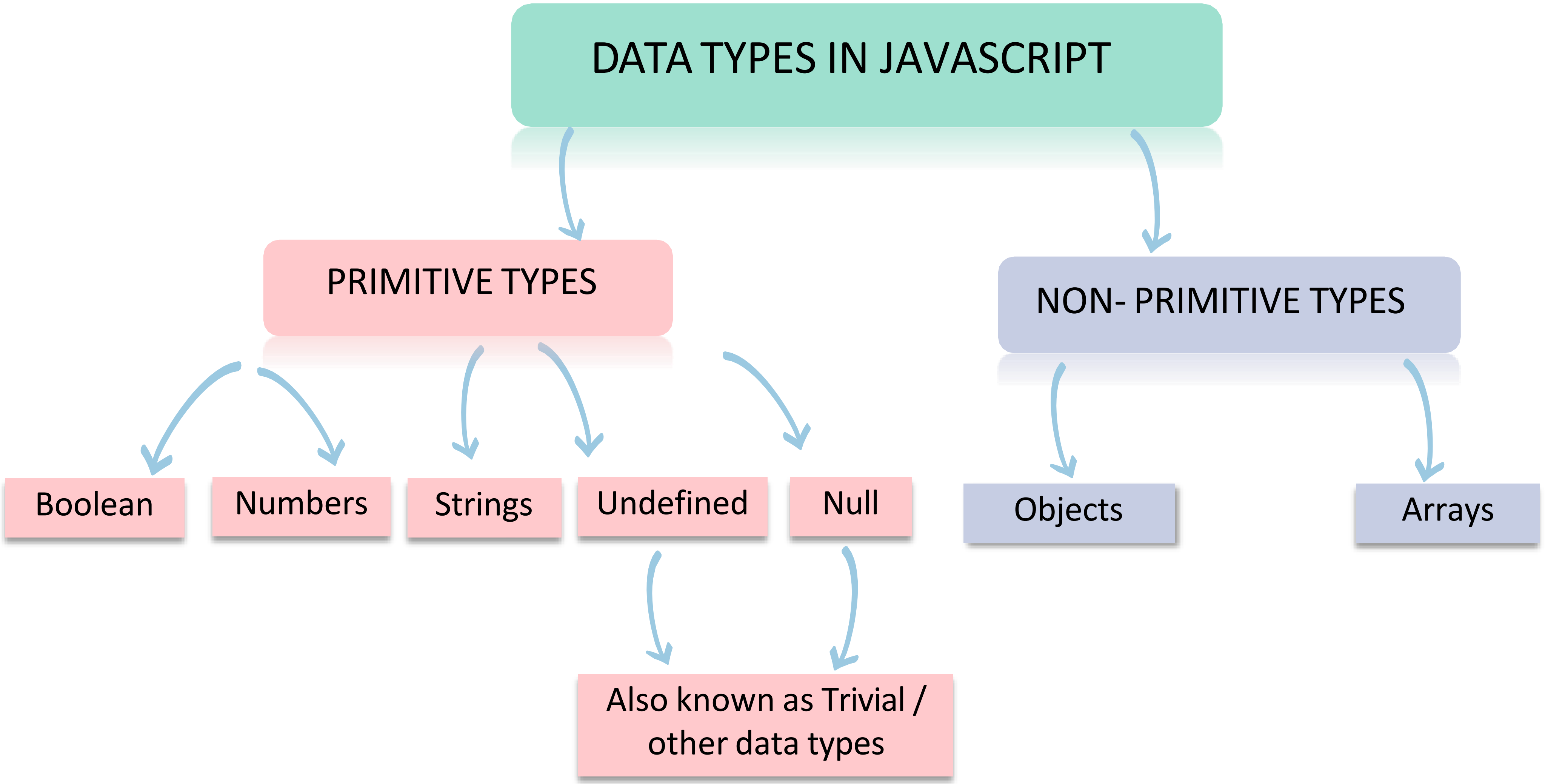
Python, PHP, Ruby, JavaScript





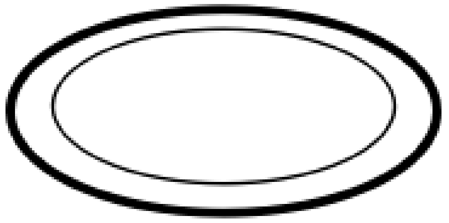
Testleaf
Always Ahead

Understanding Data Types



Null & Undefined in JS

Null



Undefined

Null & Undefined in JS

❑ **undefined: A Field Left Blank Intentionally for Later Filling**

- Imagine you're filling out an account opening form.
- There's a field labeled **"Account Number"** that **only the banker** will fill in after processing your application.
- Since you **leave it blank**, it's effectively **undefined**. The field exists, but it doesn’t have a value yet because it's waiting to be set later.

In Code Terms:

```
let accountNumber; // Declaration but no value yet, so it's undefined
console.log(accountNumber); // Output: undefined
```

❑ **null: A Field Explicitly Marked as "Not Applicable"**

- Now consider another field, **"Landline Number"**.
- Since you no longer use a landline phone, you write **"N.A."** to explicitly indicate that this field is **not applicable** to you.
- Here, you're telling the banker: *"This field is intentionally left empty because it doesn’t apply.“*

In Code Terms:

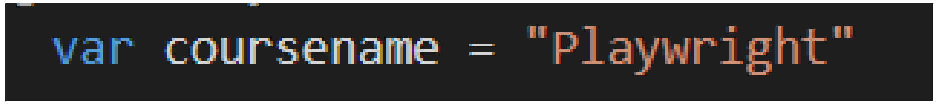
```
let landlineNumber = null; // Intentionally set to indicate no value
console.log(landlineNumber); // Output: null
```

About **variable declaration**

- ❑ Variable declaration in JavaScript refers to the process of creating a named storage space in memory, which can hold a value.

- ❑ When a variable is declared, JavaScript allocates a memory location for that variable and initializes it with a default value (e.g., undefined for uninitialized variables).

- ❑ Example



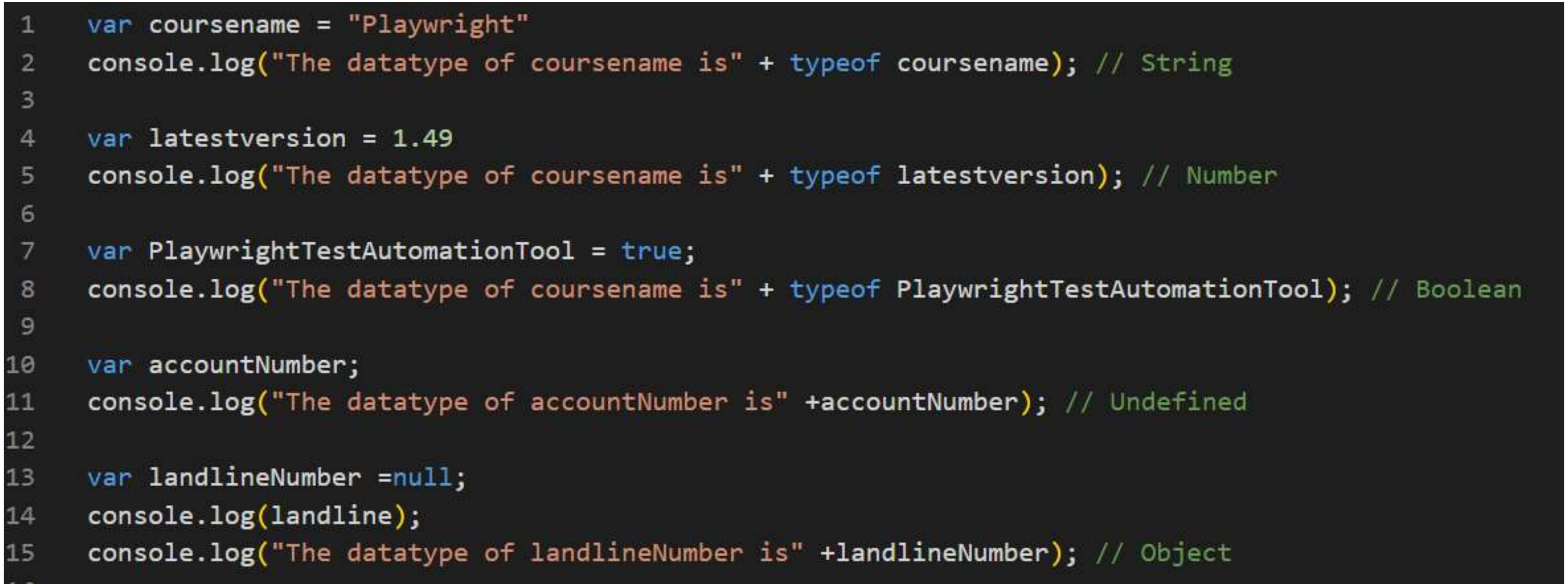
```
var coursename = "Playwright"
```

- var → variable keywords / variable declaration
- coursename → variable name
- = → assignment operator
- “ “ → Used to declare a string.
- Playwright → Value of the assigned to the variable.

About “**typeof**” operator

The **typeof** operator in JavaScript is used to determine the data type of a value or variable.
It returns a string that indicates the type of the operand.

Example :



```javascript
var coursename = "Playwright"
console.log("The datatype of coursename is" + typeof coursename); // String

var latestversion = 1.49
console.log("The datatype of coursename is" + typeof latestversion); // Number

var PlaywrightTestAutomationTool = true;
console.log("The datatype of coursename is" + typeof PlaywrightTestAutomationTool); // Boolean

var accountNumber;
console.log("The datatype of accountNumber is" +accountNumber); // Undefined

var landlineNumber =null;
console.log(landline);
console.log("The datatype of landlineNumber is" +landlineNumber); // Object
```

Why **typeof** null returns "**object**“

- Back when JavaScript was first created, values were internally represented as binary data. The type information was stored in the first few bits of the data.

- Objects had a type tag of 000 (binary).Null was also mistakenly given the same type tag 000.

- This mistake caused null to be identified as an object, and because JavaScript needed to maintain backward compatibility, this behavior was never fixed.

```
console.log(typeof null); // Output: "object"
```

Classroom on typeof operator :

Assignment Details:

Declare variables using `let` for different data types in JavaScript, and verify their types using `typeof`, including an uninitialized variable.

Assignment Requirements:

Create the following variables using let (not using var) and check their typeOf
a) firstName
b) companyName
c) mobileNumber
d) isAutomation
e) hasPlaywright (do not assign)

Print and confirm the values and data types

Hints to Solve:

Focus on initializing variables with different values, including a string, number, boolean, and leave one variable undefined to practice with `typeof`.

Expected Outcome:

Upon completion, you should be able to:
- Grasp the concepts of different data types in JavaScript

Difference between

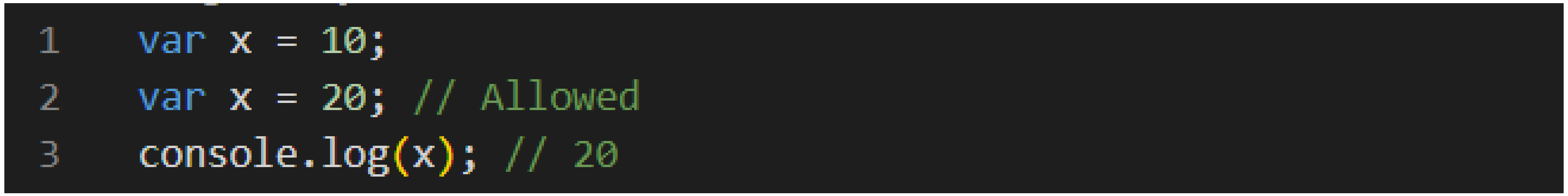
var vs let vs const

In terms of Re-declaration, Re-assignment, Scoping and Hoisting

Behavior of “var” keyword:

Redeclaration using var :

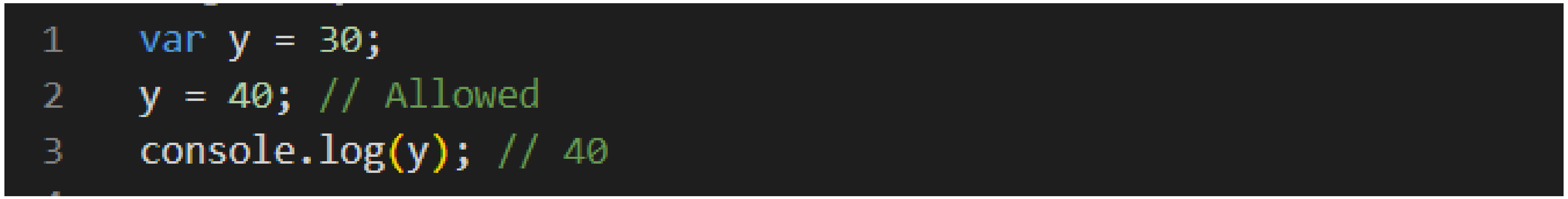
Variables declared with var can be redeclared within the same scope without throwing an error.



```
var x = 10;
var x = 20; // Allowed
console.log(x); // 20
```

Reassignment using var :

Variables declared with var can be reassigned to a new value at any time.

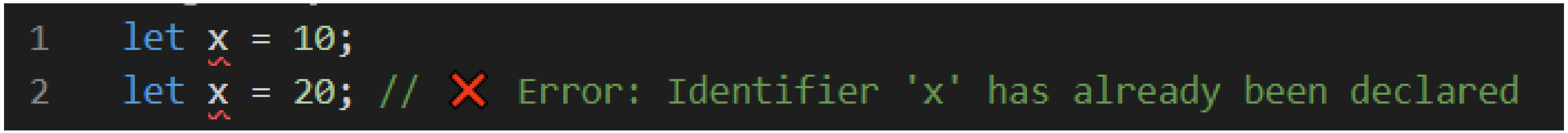


```
var y = 30;
y = 40; // Allowed
console.log(y); // 40
```

Behavior of “let” keyword:

Redeclaration in let :

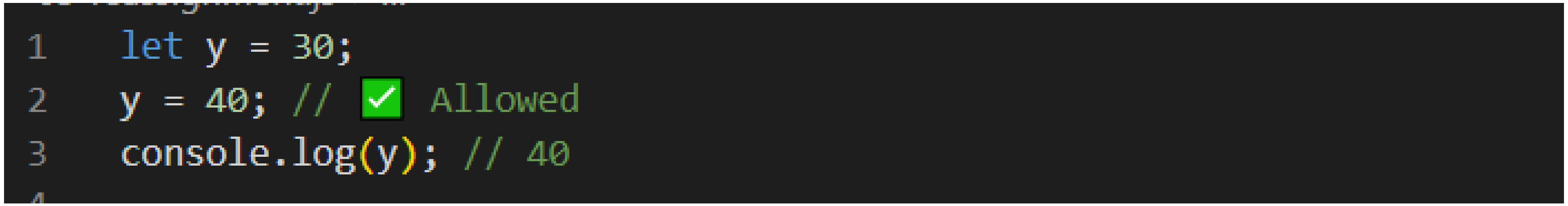
Variables declared with let cannot be redeclared in the same scope.



```
let x = 10;
let x = 20; // ❌ Error: Identifier 'x' has already been declared
```

Reassignment in let :

Variables declared with let can be reassigned new values.

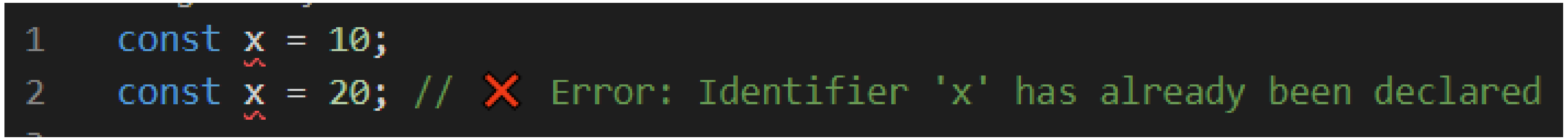


```
let y = 30;
y = 40; // ✅ Allowed
console.log(y); // 40
```

Behavior of “const” keyword:

Redeclaration in const :

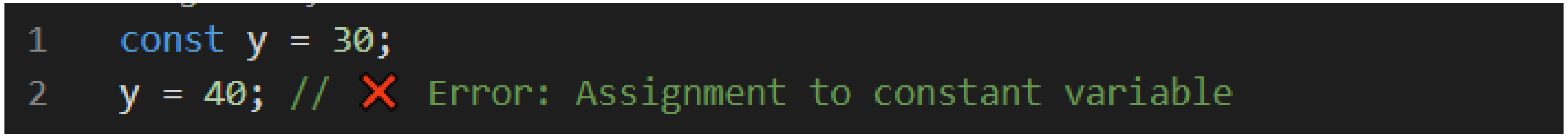
Variables declared with const cannot be redeclared in the same scope.



```
const x = 10;
const x = 20; // ❌ Error: Identifier 'x' has already been declared
```

Reassignment in const :

Variables declared with const cannot be reassigned after their initial assignment.



```
const y = 30;
y = 40; // ❌ Error: Assignment to constant variable
```

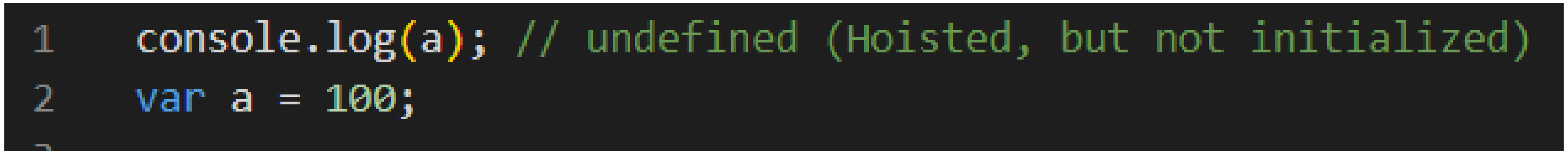
Hoisting

- Hoisting is a simple mechanism where the JavaScript interpreter moves the variable and function declarations to the top of the code block

```
console.log(x);                    // Outputs: undefined
var x = 10;
console.log(x);                    // Outputs: 10
```

Hoisting in var :

Variables declared with var are hoisted to the top of their function or global scope, but they are initialized as undefine

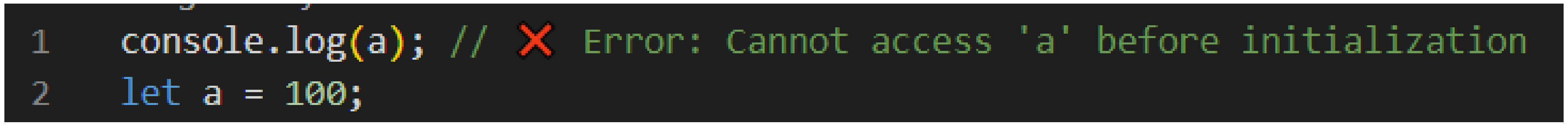


```
console.log(a); // undefined (Hoisted, but not initialized)
var a = 100;
```

Hoisting

Hoisting in let :

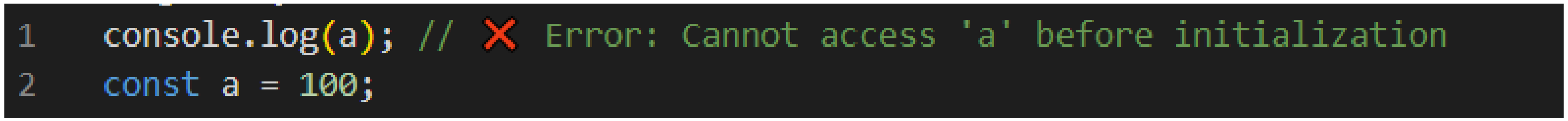
Variables declared with let are hoisted, but they are not initialized.
Accessing them before their declaration results in a ReferenceError.



```
console.log(a); // ❌ Error: Cannot access 'a' before initialization
let a = 100;
```

Hoisting in const :

Variables declared with const are hoisted, but they are not initialized.
Accessing them before their declaration results in a ReferenceError.



```
console.log(a); // ❌ Error: Cannot access 'a' before initialization
const a = 100;
```

Scoping

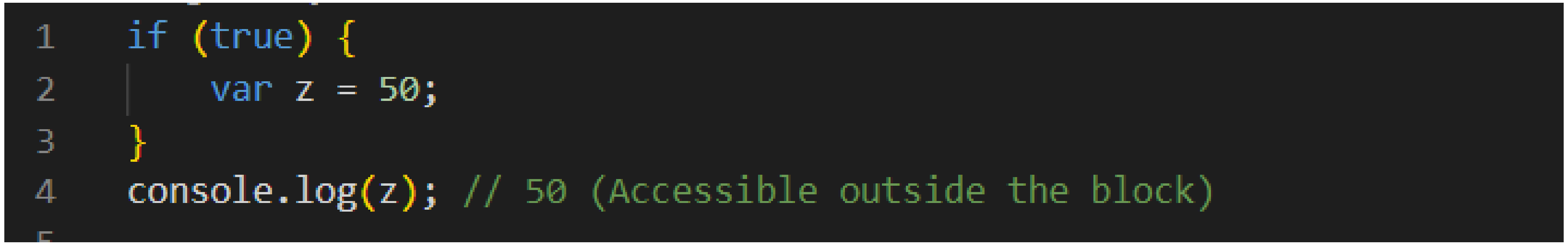
- Scoping in JavaScript refers to the ***accessibility or visibility*** of variables in different parts of the code.

- Variables in JavaScript can be scoped ***globally, functionally, or block-level,*** depending on how they are declared (var, let, or const).

Scoping in var :

✓ var is **function-scoped**, meaning it is accessible throughout the function where it is declared, even before its declaration due to hoisting.

✓ However, it is **not block-scoped**, so it can "**leak**" outside block statements like if or for.

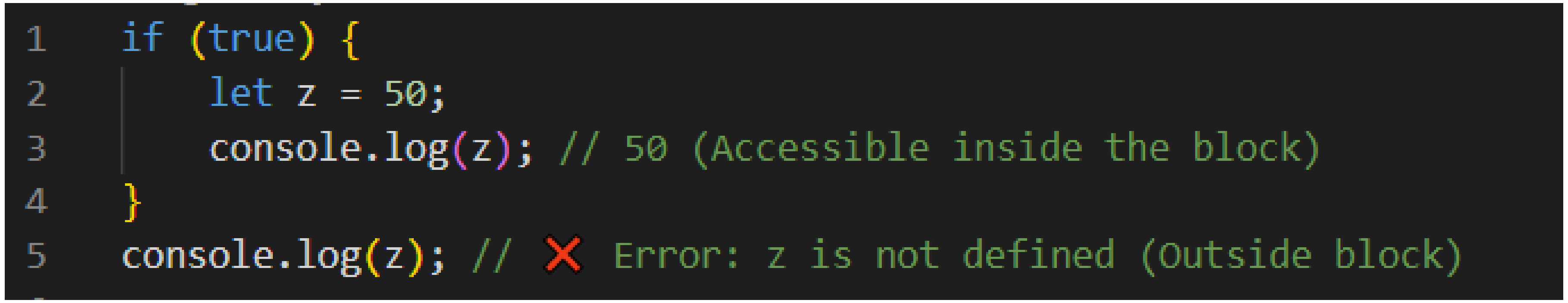


```
if (true) {
    var z = 50;
}
console.log(z); // 50 (Accessible outside the block)
```

Scoping in “let” & “const” keyword:

Scoping in let :

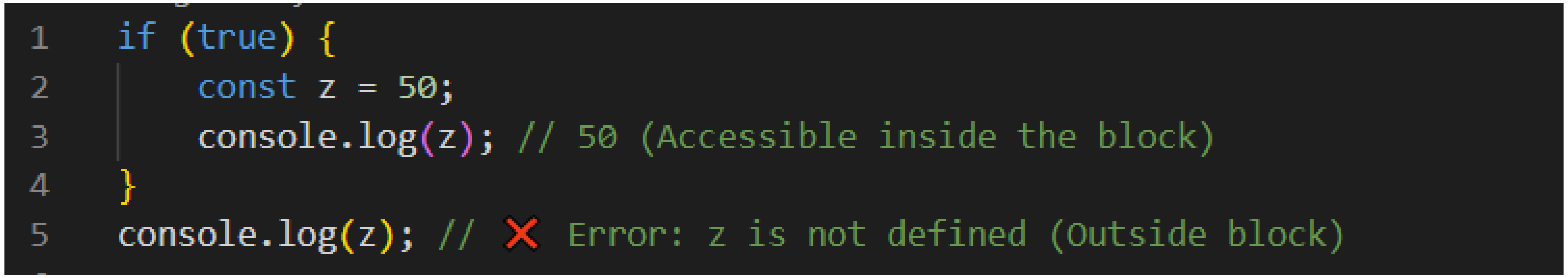
let is block-scoped, meaning it is only accessible within the block, statement, or expression where it is defined.



```
if (true) {
    let z = 50;
    console.log(z); // 50 (Accessible inside the block)
}
console.log(z); // ❌ Error: z is not defined (Outside block)
```

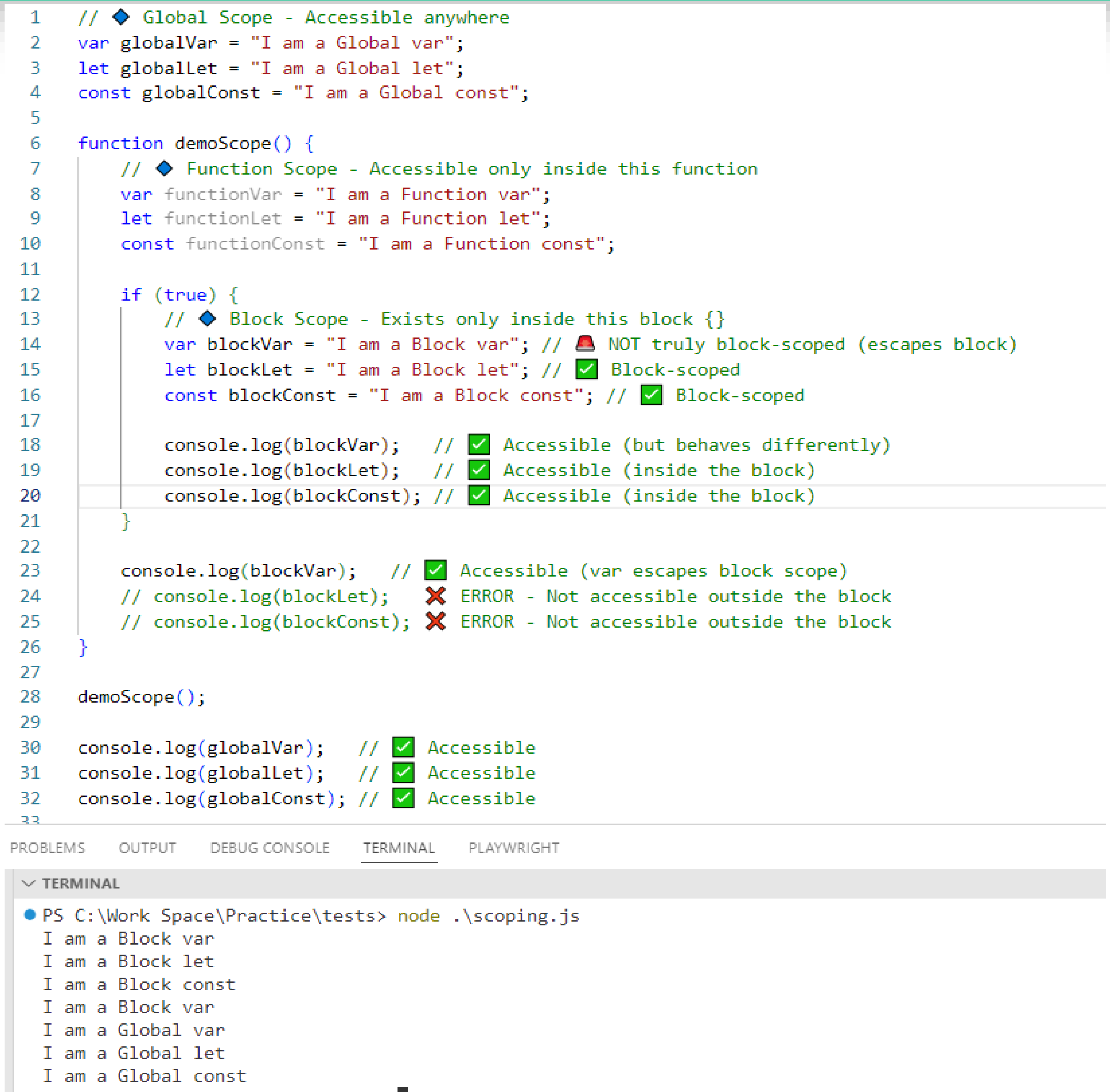
Scoping in const :

const is block-scoped, meaning it is only accessible within the block, statement, or expression where it is defined.



```
if (true) {
    const z = 50;
    console.log(z); // 50 (Accessible inside the block)
}
console.log(z); // ❌ Error: z is not defined (Outside block)
```

Scoping



```javascript
// 🔹 Global Scope - Accessible anywhere
var globalVar = "I am a Global var";
let globalLet = "I am a Global let";
const globalConst = "I am a Global const";

function demoScope() {
    // 🔹 Function Scope - Accessible only inside this function
    var functionVar = "I am a Function var";
    let functionLet = "I am a Function let";
    const functionConst = "I am a Function const";

    if (true) {
        // 🔹 Block Scope - Exists only inside this block {}
        var blockVar = "I am a Block var"; // 🚨 NOT truly block-scoped (escapes block)
        let blockLet = "I am a Block let"; // ✅ Block-scoped
        const blockConst = "I am a Block const"; // ✅ Block-scoped

        console.log(blockVar);   // ✅ Accessible (but behaves differently)
        console.log(blockLet);   // ✅ Accessible (inside the block)
        console.log(blockConst); // ✅ Accessible (inside the block)
    }

    console.log(blockVar);   // ✅ Accessible (var escapes block scope)
    // console.log(blockLet);   ❌ ERROR - Not accessible outside the block
    // console.log(blockConst); ❌ ERROR - Not accessible outside the block
}

demoScope();

console.log(globalVar);   // ✅ Accessible
console.log(globalLet);   // ✅ Accessible
console.log(globalConst); // ✅ Accessible
```

```
PS C:\Work Space\Practice\tests> node .\scoping.js
I am a Block var
I am a Block let
I am a Block const
I am a Block var
I am a Global var
I am a Global let
I am a Global const
```

Scoping sample program to understand better

Comparison Summary Table: var, let, and const

Feature	`var`	`let`	`const`
Re-declaration	✅ Allowed in the same scope	❌ Not allowed in the same scope	❌ Not allowed in the same scope
Re-assignment	✅ Allowed	✅ Allowed	❌ Not allowed
Scoping	Function-scoped or global	Block-scoped	Block-scoped
Block Awareness	❌ Does not respect block scope	✅ Respects block scope	✅ Respects block scope
Hoisting	✅ Hoisted and initialized to `undefined`	✅ Hoisted but uninitialized (**TDZ applies**)	✅ Hoisted but uninitialized (**TDZ applies**)
TDZ (Temporal Dead Zone)	❌ No TDZ	✅ Exists (cannot access before declaration)	✅ Exists (cannot access before declaration)

Testleaf
Always Ahead

Classroom on keyword var, let, const behavior :

Assignment Details:

Declare a global variable and shadow it inside a function using both `var` and `let` to see how they behave differently when printed.

Assignment Requirements:

1. Declare a const name as browserVersion (global)
2. Assign value as Chrome
3. Create a function by name getBrowserVersion
4. Create if condition inside function to check if browser is chrome, then
5. Declare a local variable (browserVersion) and print that variable inside function (outside block)
6. Call that function from the javascript

Hints to Solve:

- Use 'var' first as block variable and then convert that as 'let'
- Confirm how it works

Expected Outcome:

Upon completion, you should be able to:

- Understand the concepts of var, let and const and the hoisting principles

Testleaf
Always Ahead