

TypeScript Basics for Automation Testers – Day 3

Topic: Variables in TypeScript (Part 2 – Understanding Scopes, Re-declaration & Hoisting)

In the previous part, we learned about the three keywords used to declare variables in TypeScript — **var**, **let**, and **const**.

In this part, we'll understand **five aspects** that differentiate them clearly.

Five Aspects that Differentiate var, let, and const

1. Scope of the variable
 2. Declaration or value assignment
 3. Re-declaration
 4. Re-initialization or re-assignment
 5. Hoisting
-

1. Scope of the Variable

What does “scope” mean?

Scope means the *area where something is accessible or valid*.

Example: You can access your office Wi-Fi only inside the office premises — that's its **scope**.

In Java:

There are 3 types of scope:

1. **Local scope** – accessible only inside the method or block.
2. **Instance scope** – tied to an object (non-static).
3. **Static scope** – shared across all objects.

In TypeScript/JavaScript:

We only have 2 types of scope:

1. **Functional Scope**
 2. **Block Scope**
-

Functional Scope

Meaning: Variables declared inside a function are accessible anywhere *within* that function.

Why? Because `var` keyword is function-scoped.

Example – var keyword

```
function varScope() {  
  if (true) {  
    var msg = "Hello Yogi";  
    console.log(msg); // accessible inside the block  
  }  
  console.log(msg); // still accessible here (inside function)  
}  
varScope();
```



```
TS functionalScopeExample.ts ×  
day2 > TS functionalScopeExample.ts > ...  
1  function varScope() {  
2    if (true) {  
3      var msg = "Hello Yogi";  
4      console.log(msg); // accessible inside the block  
5    }  
6    console.log(msg); // still accessible here (inside function)  
7  }  
8  varScope();  
-  
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  PLAYWRIGHT  
● [redacted] TSDemo % tsx day2/functionalScopeExample.ts  
Hello Yogi  
Hello Yogi
```

Practice Tip:

Try printing `msg` *outside* the function — it will throw an error because it's not accessible globally.



```
TS functionalScopeExample.ts 1 ×  
day2 > TS functionalScopeExample.ts > ...  
1  function varScope() {  
2    if (true) {  
3      var msg = "Hello Yogi";  
4      console.log(msg); // accessible inside the block  
5    }  
6    console.log(msg); // still accessible here (inside function)  
7  }  
8  console.log(msg);  
9  varScope();  
10
```

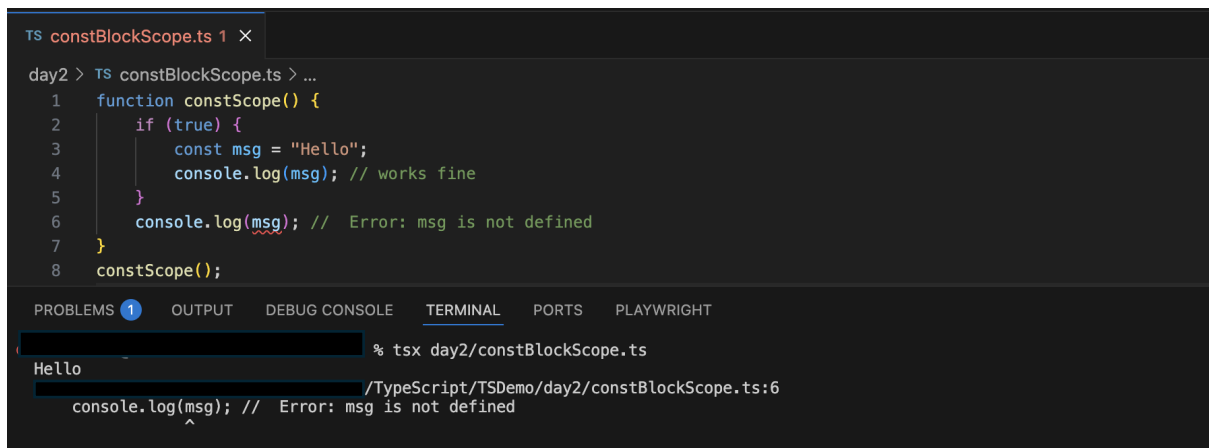
Block Scope

Meaning: A variable declared inside `{ }` is only accessible within those braces (block).

`let` and `const` are **block scoped**, meaning they live and die inside the `{ }` block where they are declared.

Example 1 – const (Block Scope)

```
function constScope() {  
  if (true) {  
    const msg = "Hello";  
    console.log(msg); // works fine  
  }  
  console.log(msg); // Error: msg is not defined  
}  
constScope();
```



The screenshot shows a VS Code editor with a file named `constBlockScope.ts`. The code in the file is identical to the one in the previous block. The terminal output shows the command `% tsx day2/constBlockScope.ts` being executed, which prints `Hello` to the console. Below the output, the terminal shows the error message `/TypeScript/TSDemo/day2/constBlockScope.ts:6 console.log(msg); // Error: msg is not defined` with a red squiggly line under the `msg` variable, indicating the error.

Example 2 – let (Block Scope)

```
function letScope() {  
  for (let i: number = 0; i <= 10; i++) {  
    console.log(i); // works fine  
  }  
  console.log(i); // Error: i is not defined  
}  
letScope();
```

```
TS letBlockScope.ts 1 X
day2 > TS letBlockScope.ts > letScope
1 function letScope(){
2   for(let i:number=1;i<=10;i++){
3     console.log(i);
4   }
5   console.log(i);
6 }
7 letScope();
```

Example 3 – Scope Difference

```
function scopeDifference() {
  if (true) {
    var num1 = 10;
    let num2 = 20;
    const num3 = 30;
    console.log(num1, num2, num3); // all accessible here
  }
  console.log(num1); // works
  console.log(num2); // Error
  console.log(num3); // Error
}
scopeDifference();
```

```
TS scopeDifference.ts 2 X
day2 > TS scopeDifference.ts > scopeDifference
1 function scopeDifference(){
2   if(true){
3     var num1:number=10;
4     let num2:number=20;
5     const num3:number=30;
6     console.log(num1,num2,num3);
7   }
8   console.log(num1);//var is accessible outside the block
9   console.log(num2);//let is not accessible outside the block
10  console.log(num3);//const is not accessible outside the block
11 }
12 scopeDifference();
```

2. Declaration / Value Assignment

Declaration means *introducing* something (creating a name),
Assignment means *giving a value* to it.

Example 1 – var

```
var variableName;  
console.log(variableName); // undefined (declared but not initialized)  
variableName = "Yogi";  
console.log(variableName); // Yogi
```




```
TS declaration.ts X  
day2 > TS declaration.ts > ...  
1   var variableName;  
2   console.log(variableName); // undefined (declared but not initialized)  
  
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  PLAYWRIGHT  
● [redacted] -MAC TSDemo % tsx day2/declaration.ts  
undefined
```

Note: If you don't specify a datatype, TypeScript assumes it as any.

Example 2 – let

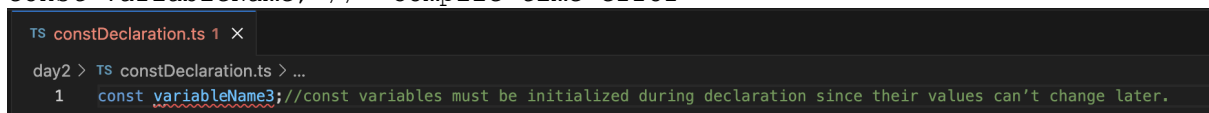
```
let variableName;  
console.log(variableName); // undefined  
variableName = "Yogi";  
console.log(variableName);
```



```
TS letDeclaration.ts X  
day2 > TS letDeclaration.ts > ...  
1   let variableName2;  
2   console.log(variableName2);  
  
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  PLAYWRIGHT  
● [redacted] -MAC TSDemo % tsx day2/letDeclaration.ts  
undefined
```

Example 3 – const

```
const variableName; // Compile-time error
```



```
TS constDeclaration.ts 1 X  
day2 > TS constDeclaration.ts > ...  
1   const variableName3; //const variables must be initialized during declaration since their values can't change later.
```

Explanation:

const variables must be initialized during declaration since their values can't change later.

3. Re-declaration

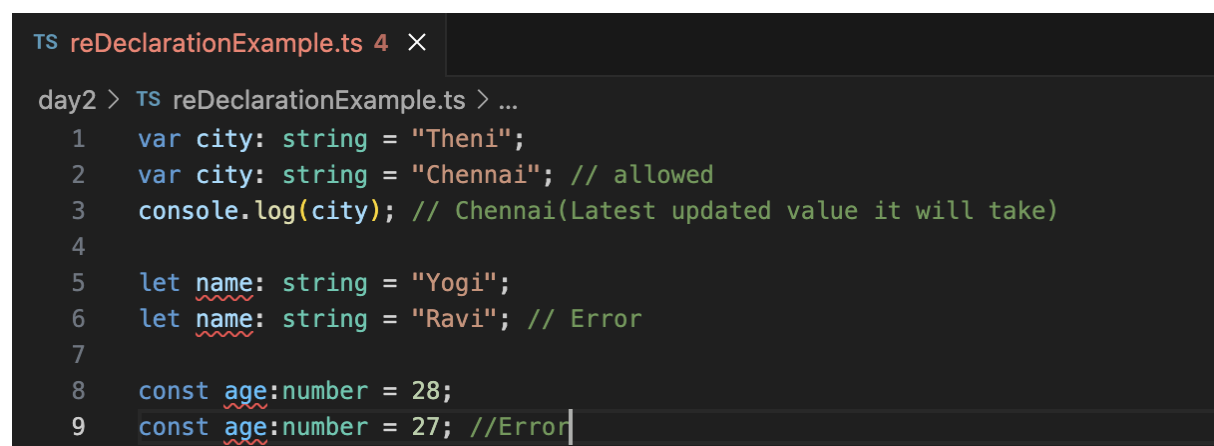
Meaning: Declaring something again with the same name.

In programming: Declaring a variable with the same name again in the same scope.

- `var` → Allows re-declaration
- `let` and `const` → Do not allow re-declaration

Example

```
var city: string = "Theni";  
var city: string = "Chennai"; // allowed  
console.log(city); // Chennai  
  
let name: string = "Yogi";  
let name: string = "Ravi"; // Error
```



The screenshot shows a VS Code editor window titled "TS reDeclarationExample.ts 4 X". The editor content is as follows:

```
day2 > TS reDeclarationExample.ts > ...  
1  var city: string = "Theni";  
2  var city: string = "Chennai"; // allowed  
3  console.log(city); // Chennai(Latest updated value it will take)  
4  
5  let name: string = "Yogi";  
6  let name: string = "Ravi"; // Error  
7  
8  const age:number = 28;  
9  const age:number = 27; //Error|
```

The code demonstrates that `var` allows re-declaration, while `let` and `const` result in TypeScript errors. The `console.log` statement outputs "Chennai" because it prints the latest value of the `city` variable.

Reason: Allowing re-declaration breaks type safety — one of the main reasons we prefer `let` or `const`.

4. Re-initialization / Re-assignment

Meaning: Assigning a new value to an already declared variable.

Initialization:

```
TS initializationExample.ts X
day2 > TS initializationExample.ts > ...
1  var weight:number = 74;
2  let height:number = 178;
3  const gender:string = "male";
4  console.log(weight, height,gender);

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  PLAYWRIGHT
● -MAC TSDemo % tsx day2/initializationExample.ts
74 178 male
```

Re-Initialization:

```
TS reinitializationExample.ts 1 X
day2 > TS reinitializationExample.ts > ...
1  var myweight:number = 74;
2  let myheight:number = 178;
3  const mygender:string = "male";
4  console.log(myweight, myheight,mygender);
5  myweight=78;//Var allows to reinitialize
6  myheight=180;//let allows to reinitialize
7  mygender="female";//const never allows to reinitialize
```

Example 1 – var

```
var name: string = "Yogi";
name = "Thangam"; // allowed
console.log(name); // Thangam
```

Example 2 – let

```
let name: string = "Yogi";
name = "Thangam"; // allowed
console.log(name);
```

Example 3 – const

```
const character: string = "Nallavan";
character = "Kettavan"; // Error: Assignment to constant variable
```

Note: const values are constant — cannot be reassigned.

5. Hoisting

“Hoist” means *to lift up or raise*.

In programming:

Hoisting means *variable declarations are moved to the top of their scope before code execution*.

Example 1 – var

```
console.log(future); // undefined
var future: string = "surprise";
console.log(future);
```

Explanation:

With `var`, the declaration gets hoisted but not the value, so it prints `undefined`.

Example 2 – let

```
console.log(future); // Error: Cannot access before initialization
let future: string = "surprise";
```

Example 3 – const

```
console.log(future); // Error: Cannot access before initialization
const future: string = "surprise";
```



```
TS hoistingExample.ts 5 X
day2 > TS hoistingExample.ts > ...
1 //Example 1 - var
2 console.log(future3); // undefined
3 var future3: string = "surprise";
4 console.log(future3);
5 //Explanation:
6 //With var, the declaration gets hoisted but not the value, so it prints undefined.
7
8 //Example 2 - let
9 console.log(future1); // Error: Cannot access before initialization
10 let future1: string = "surprise";
11 console.log(future1);
12 //Example 3 - const
13 console.log(future2); // Error: Cannot access before initialization
14 const future2: string = "surprise";
15 console.log(future2);
16

PROBLEMS 5 OUTPUT DEBUG CONSOLE TERMINAL PORTS PLAYWRIGHT
TSDemo % tsx day2/hoistingExample.ts
undefined
surprise
Learning/TypeScript/TSDemo/day2/hoistingExample.ts:9
console.log(future1); // Error: Cannot access before initialization
           ^
ReferenceError: Cannot access 'future1' before initialization
```

Why avoid var?

Because of hoisting, it can create bugs — code may behave differently than expected.

Practice Summary

Keyword	Scope	Re-declare	Re-initialize	Must Initialize	Hoisted
var	Function	Yes	Yes	No	Yes (undefined)
let	Block	No	Yes	No	No
const	Block	No	No	Yes	No

Questions

1. What are the five aspects that differentiate var, let, and const?
2. Which keyword is function-scoped in TypeScript?
3. Why is var not recommended for modern TypeScript projects?
4. What is the main difference between block and function scope?
5. Which keywords allow re-initialization?
6. Which keyword must be initialized during declaration?
7. What is hoisting in TypeScript?

8. Why are `let` and `const` safer to use than `var`?

Answers

A1. The five aspects that differentiate `var`, `let`, and `const` are:

1. Scope of the variable
2. Declaration or value assignment
3. Re-declaration
4. Re-initialization (or re-assignment)
5. Hoisting

A2. `var` is **function-scoped** in TypeScript. It means the variable declared with `var` can be accessed anywhere inside the function, even outside blocks like `if` or `for`.

A3. `var` is not recommended because it has **function scope** and supports **re-declaration and hoisting**, which can cause unexpected behavior and make debugging harder.

A4. The main difference is:

- **Function Scope:** Variables can be accessed anywhere inside the function.
- **Block Scope:** Variables are accessible only within the block `{ }` where they are declared.

A5. Both `var` and `let` allow re-initialization (changing the value after declaration). `const` does not allow re-initialization.

A6. `const` must be initialized at the time of declaration, otherwise TypeScript will throw a compile-time error.

A7. Hoisting means moving variable declarations to the top of their scope before code execution.

With `var`, this happens silently — undeclared variables return `undefined`.

With `let` and `const`, accessing variables before declaration causes a compile-time error.

A8. `let` and `const` are safer because they are **block-scoped**, prevent **accidental re-declaration**, and throw errors for **illegal access before initialization**, ensuring type safety and cleaner code.
