Variables

Week	3

> Adib Sakhawat
> IUT CSE '21

Variables are fundamental in programming languages, acting as containers for storing data values. In JavaScript, variables can be declared using `var`, `let`, or `const`. Understanding the differences between these declarations is crucial for writing efficient and error-free code.

Declaring Variables

Using `var`

The `var` keyword is the oldest way to declare variables in JavaScript. Variables declared with `var` are function-scoped and can be re-declared and updated within their scope.

```
var x = 10;
var y = 'Hello, World!';
```

Using `let`

Introduced in ES6 (ECMAScript 2015), `let` allows you to declare block-scoped variables. Variables declared with `let` can be updated but not re-declared within the same scope.

```
let x = 10;
let y = 'Hello, World!';
```

Using `const`

Also introduced in ES6, `const` declares block-scoped variables whose values cannot be reassigned. Variables declared with `const` must be initialized at the time of declaration.

```
const PI = 3.14159;
const greeting = 'Hello, World!';
```

Scope Differences

Scope determines the accessibility of variables in different parts of your code. JavaScript has three types of scopes:

1. **Global Scope**
2. **Function Scope**
3. **Block Scope**

Global Scope

A variable declared outside all functions and blocks has a global scope. It can be accessed from anywhere in the code.

```
var globalVar = 'I am global';

function checkGlobalScope() {
    console.log(globalVar); // Outputs: I am global
}

checkGlobalScope();
console.log(globalVar); // Outputs: I am global
```

Function Scope (`var`)

Variables declared with `var` inside a function are function-scoped. They are accessible within the function but not outside of it.

```
function varFunctionScope() {
    var functionVar = 'I am function scoped';
    console.log(functionVar); // Outputs: I am function sco
ped
}

varFunctionScope();
```

```
console.log(functionVar); // ReferenceError: functionVar is
not defined
```

Block Scope (`let` and `const`)

Variables declared with `let` or `const` are block-scoped, meaning they are only accessible within the nearest set of curly braces `{}`.

```
if (true) {
    let blockLet = 'I am block scoped';
    const blockConst = 'I am also block scoped';
    console.log(blockLet);   // Outputs: I am block scoped
    console.log(blockConst); // Outputs: I am also block sc
oped
}

console.log(blockLet);   // ReferenceError: blockLet is not
defined
console.log(blockConst); // ReferenceError: blockConst is n
ot defined
```

Hoisting

Hoisting is JavaScript's default behavior of moving variable and function declarations to the top of their containing scope before code execution.

Hoisting with `var`

Variables declared with `var` are hoisted and initialized with `undefined`.

```
console.log(hoistedVar); // Outputs: undefined
var hoistedVar = 'I am hoisted';
console.log(hoistedVar); // Outputs: I am hoisted
```

Under the hood, JavaScript interprets the code as:

```
var hoistedVar;
console.log(hoistedVar); // Outputs: undefined
```

```
hoistedVar = 'I am hoisted';
console.log(hoistedVar); // Outputs: I am hoisted
```

Hoisting with `let` and `const`

Variables declared with `let` and `const` are hoisted but not initialized. Accessing them before their declaration results in a **ReferenceError** due to the **Temporal Dead Zone (TDZ)**.

```
console.log(hoistedLet); // ReferenceError: Cannot access 'hoistedLet' before initialization
let hoistedLet = 'I am not hoisted';
```

```
console.log(hoistedConst); // ReferenceError: Cannot access 'hoistedConst' before initialization
const hoistedConst = 'I am not hoisted';
```

Code Examples Illustrating Each Concept

Example with `var`

```
function varExample() {
    var x = 1;
    if (true) {
        var x = 2; // Same variable!
        console.log(x); // Outputs: 2
    }
    console.log(x); // Outputs: 2
}

varExample();
```

Since `var` is function-scoped, the variable `x` inside the `if` block overwrites the `x` declared in the function.

Example with `let`

```
function letExample() {
    let x = 1;
    if (true) {
        let x = 2; // Different variable
        console.log(x); // Outputs: 2
    }
    console.log(x); // Outputs: 1
}

letExample();
```

With `let`, the variable `x` inside the `if` block is a different variable, due to block scoping.

Example with `const`

```
const y = 5;
y = 10; // TypeError: Assignment to constant variable.
```

Variables declared with `const` cannot be reassigned.

Hoisting with `var`

```
function hoistVar() {
    console.log(msg); // Outputs: undefined
    var msg = 'Hoisted';
    console.log(msg); // Outputs: Hoisted
}

hoistVar();
```

Hoisting with `let` and `const`

```
function hoistLet() {
    console.log(msg); // ReferenceError: Cannot access 'msg' before initialization
    let msg = 'Not hoisted';
}

hoistLet();
```

Best Practices

- **Prefer `let` and `const` over `var`**: They provide block scoping, which aligns more closely with other programming languages and reduces unexpected behavior.
- **Use `const` when possible**: If you don't need to reassign a variable, use `const` to prevent accidental reassignments.
- **Avoid Hoisting Issues**: Declare all variables at the top of their scope to make the code more predictable.
- **Be Mindful of Scope**: Understand the scope in which your variables exist to prevent unintended side effects.

Summary

- `var`: Function-scoped, hoisted and initialized with `undefined`, can be re-declared and updated.
- `let`: Block-scoped, hoisted but not initialized (TDZ applies), can be updated but not re-declared in the same scope.
- `const`: Block-scoped, hoisted but not initialized (TDZ applies), cannot be updated or re-declared.