What is the difference between `var`, `let`, and `const`?

Introduction: Prior to ES6 (ECMAScript 2015), `var` was the only way to declare variables in JavaScript. ES6 introduced `let` and `const`, which brought significant improvements to variable declaration regarding scoping, hoisting, and mutability. Understanding these differences is crucial for writing modern, predictable, and bug-free JavaScript code.

Core Concepts: `var` vs. `let` vs. `const`

> The primary differences between `var`, `let`, and `const` revolve around their **scope**, **hoisting behavior**, and whether they can be **redeclared** or **reassigned**.

1. `var` Keyword (Pre-ES6)

- **Scope: Function-scoped or Global-scoped.**
  Variables declared with `var` are scoped to the nearest function block or, if declared outside any function, they become global variables. They ignore block scopes (like `if` statements, `for` loops, or plain curly braces `{}`).

`var` Scope Example

```javascript
function exampleVarScope() {
    var x = 10; // Function-scoped
    if (true) {
        var y = 20; // Also function-scoped, not block-scoped
        console.log(x); // 10
        console.log(y); // 20
    }
```

```
    console.log(x); // 10 (accessible)
    console.log(y); // 20 (accessible even outside the if block)
}
exampleVarScope();
// console.log(x); // ReferenceError: x is not defined (outside function
scope)
```

- **Hoisting: Declarations are hoisted, initializations are not.**
  `var` declarations are "hoisted" to the top of their function or global scope. This means you can use a `var` variable before its actual declaration in the code, but its value will be `undefined` until the line of assignment is reached.

  ### `var` Hoisting Example

  ```
  JAVASCRIPT
  console.log(myVar); // Output: undefined (declaration hoisted, assignment
  is not)
  var myVar = 'I am var';
  console.log(myVar); // Output: I am var
  ```

- **Redeclaration & Reassignment: Both allowed.**
  You can redeclare a `var` variable multiple times within the same scope without an error, and you can reassign its value.

  ### `var` Redeclaration/Reassignment

  ```
  JAVASCRIPT
  var a = 5;
  console.log(a); // 5
  var a = 10; // Redeclared, no error
  console.log(a); // 10
  a = 15; // Reassigned
  console.log(a); // 15
  ```

2. `let` Keyword (ES6+)

- **Scope: Block-scoped.**

Variables declared with `let` are scoped to the block (curly braces `{}`) in which they are defined. This includes `if` statements, `for` loops, and any custom block.

`let` Scope Example

```javascript
function exampleLetScope() {
    let x = 10;
    if (true) {
        let y = 20; // Block-scoped to this if block
        console.log(x); // 10
        console.log(y); // 20
    }
    console.log(x); // 10 (accessible)
    // console.log(y); // ReferenceError: y is not defined (outside its block scope)
}
exampleLetScope();
```

- **Hoisting: Hoisted into Temporal Dead Zone (TDZ).**
  `let` declarations are hoisted to the top of their block, but they are not initialized. Instead, they enter a "Temporal Dead Zone" (TDZ) from the beginning of their scope until their declaration is processed. Accessing a `let` variable within its TDZ will throw a `ReferenceError`.

`let` Hoisting Example

```javascript
// console.log(myLet); // ReferenceError: Cannot access 'myLet' before initialization (TDZ)
let myLet = 'I am let';
console.log(myLet); // Output: I am let
```

- **Redeclaration & Reassignment: Cannot be redeclared, can be reassigned.**
  You cannot redeclare a `let` variable in the same scope. However, you can reassign its value.

`let` Redeclaration/Reassignment

```javascript
let b = 5;
console.log(b); // 5
// let b = 10; // SyntaxError: 'b' has already been declared (cannot redeclare)
b = 15; // Reassignment is allowed
console.log(b); // 15
```

3. `const` Keyword (ES6+)

- **Scope: Block-scoped.**
  Similar to `let`, `const` variables are block-scoped.
- **Hoisting: Hoisted into Temporal Dead Zone (TDZ).**
  Like `let`, `const` declarations are hoisted into the TDZ. They must be initialized at the time of declaration; otherwise, a `SyntaxError` occurs.

`const` Hoisting Example

```javascript
// console.log(myConst); // ReferenceError: Cannot access 'myConst' before initialization (TDZ)
// const myConst; // SyntaxError: Missing initializer in const declaration
const myConst = 'I am const';
console.log(myConst); // Output: I am const
```

- **Redeclaration & Reassignment: Neither allowed.**
  `const` variables cannot be redeclared, nor can their value be reassigned after initial declaration. This makes them ideal for values that should not change.

`const` Redeclaration/Reassignment

```javascript
const c = 5;
console.log(c); // 5
```

```
// const c = 10; // SyntaxError: 'c' has already been declared
// c = 15; // TypeError: Assignment to constant variable.
```

Important Note on `const` with Objects/Arrays:
While a `const` variable itself cannot be reassigned, if its value is an object or an array, the *contents* of that object or array *can* be modified (mutated).

```javascript
const myObject = { name: 'Alice' };
myObject.name = 'Bob'; // Allowed: modifying object property
console.log(myObject); // { name: 'Bob' }

// myObject = { name: 'Charlie' }; // TypeError: Assignment to constant variable. (Not allowed: reassigning the variable itself)

const myArray = [1, 2];
myArray.push(3); // Allowed: modifying array content
console.log(myArray); // [1, 2, 3]
```

Why It Matters: Importance and Use Cases

The introduction of `let` and `const` was a significant improvement for JavaScript, making code more predictable and easier to reason about:

- **Reduced Bugs:** Block-scoping helps prevent variable overwriting and unexpected behavior, especially in loops and conditional blocks.
- **Improved Readability:** `const` clearly signals that a variable's reference should not change, aiding in code understanding.
- **Predictable Scope:** Moving from function-scoping (`var`) to block-scoping (`let`, `const`) aligns JavaScript more closely with other C-like languages and reduces common pitfalls.
- **Temporal Dead Zone (TDZ):** TDZ errors for `let`/`const` provide immediate feedback for accessing variables before declaration, promoting better coding practices than `undefined` from `var` hoisting.

Real-world Scenario: In a loop, if you use `var` for the loop counter in an asynchronous callback, all callbacks might refer to the same final value of the counter. With `let`, each iteration has its own block-scoped variable, solving this common closure issue.

```javascript
// Common var issue in loops with setTimeout
for (var i = 0; i < 3; i++) {
    setTimeout(function() {
        console.log(i); // Will log 3, three times
    }, 100 * i);
}

// Solution with let
for (let j = 0; j < 3; j++) {
    setTimeout(function() {
        console.log(j); // Will log 0, 1, 2
    }, 100 * j);
}
```

Best Practices

- **Prefer `const` by Default:** Use `const` for any variable whose value (or reference, for objects/arrays) should not be reassigned. This signals intent and helps prevent accidental modifications.
- **Use `let` for Reassignable Variables:** Only use `let` when you explicitly know that the variable's value needs to change over its lifetime within its scope.
- **Avoid `var`:** In modern JavaScript development, `var` is largely obsolete. Its function-scoping and permissive redeclaration behavior can lead to bugs that are harder to track.
- **Declare Variables at Top of Scope:** Even with block-scoping, declaring variables at the top of their logical block improves readability.

Common Mistakes

- **Using `var` in Loops with Closures:** As demonstrated above, this is a classic source of bugs.

- **Forgetting `const` for Objects/Arrays Allows Mutation:** Developers sometimes mistakenly believe `const` makes objects/arrays immutable, forgetting that only the variable's *reference* is constant, not its contents.
- **Accessing `let`/`const` before Declaration:** This leads to `ReferenceError` due to the TDZ, which can be confusing if one expects `undefined` like with `var`.
- **Redeclaring `let`/`const` in Same Scope:** This will throw a `SyntaxError`.

Performance Impact

The performance implications are generally minor and often negligible for typical applications, but there are nuances:

- **Optimization:** Modern JavaScript engines can sometimes optimize code using `const` more efficiently because they know the variable's reference won't change. This can lead to minor performance gains, especially in highly optimized code paths.
- **Predictability:** The more predictable scope and immutability offered by `let` and `const` can indirectly lead to more maintainable and less bug-prone code, which in turn can lead to fewer performance issues caused by logical errors.
- **TDZ Overhead (minimal):** The runtime mechanism for the Temporal Dead Zone in `let` and `const` has a very slight, almost unnoticeable, overhead compared to `var`, but this is overwhelmingly outweighed by the benefits of better scoping.

> **Key takeaway:** Embrace `let` and `const` for clearer, safer, and more maintainable JavaScript code. Use `const` as your default, and switch to `let` only when reassignment is necessary. Avoid `var` in new code.

For more educational content and insights -

Connect with Sumedh