JavaScript offers three keywords for declaring variables: `var`, `let`, and `const`. Each has its own characteristics and use cases. Understanding the differences is crucial for writing correct and maintainable JavaScript code.

1. `var`:

- **Function Scope:** Variables declared with `var` have function scope. This means they are only accessible within the function where they are declared and any nested functions within that function. If declared outside of any function, they have global scope.
- **Hoisting:** `var` declarations are hoisted to the top of their scope. This means the variable is declared even before the line where you write it in your code. However, the initialization (assignment of a value) happens only when the line is executed. This can lead to unexpected behavior.
- **Redeclaration and Reassignment:** You can redeclare and reassign `var` variables within the same scope.

```
function myFunction() {
var x = 10; // Function scope

if (true) {
    var x = 20; // Redeclared x within the same scope
    console.log(x); // Output: 20
}

console.log(x); // Output: 20 (x was redeclared)
}

myFunction();

console.log(x); // Error: x is not defined (function scope)

var y = 30; // Global scope
console.log(y); // Output: 30

var y = 40; // Redeclared y
console.log(y); // Output: 40
```

2. `let`:

- **Block Scope:** Variables declared with `let` have block scope. This means they are only accessible within the block (e.g., within an `if` statement, `for` loop, or just a `{}`

block) where they are defined and any nested blocks. This is a significant improvement over `var` because it provides more control over variable visibility.

- **Hoisting (Temporal Dead Zone):** `let` declarations are hoisted, but they are not initialized. Trying to access a `let` variable before its declaration results in a `ReferenceError` (Temporal Dead Zone). This helps prevent some of the problems associated with `var` hoisting.
- **Reassignment:** You can reassign `let` variables within their scope.
- **No Redeclaration:** You cannot redeclare a `let` variable within the same scope.

```
function myFunction() {
     let x = 10; // Block scope

     if (true) {
         let x = 20; // A different x in a nested block
         console.log(x); // Output: 20
     }

     console.log(x); // Output: 10 (the original x)
     }

     myFunction();

     console.log(x); // Error: x is not defined (block scope)


     let y = 30;
     y = 40; // Reassignment is allowed
     console.log(y); // Output: 40

     let y = 50; // Error: Identifier 'y' has already been declared
```

3. `const`:

- **Block Scope:** Like `let`, `const` variables also have block scope.
- **Hoisting (Temporal Dead Zone):** `const` declarations are also hoisted but not initialized, resulting in a Temporal Dead Zone.
- **No Redeclaration:** You cannot redeclare a `const` variable within the same scope.
- **No Reassignment (Mostly):** `const` variables cannot be reassigned. This means you cannot change the value of a `const` variable after it has been initialized. *However*, if

the `const` variable holds an object or an array, you *can* modify the *properties* of the object or the *elements* of the array. The `const` variable itself will still point to the same object or array in memory, but the contents of that object or array can be changed.

-

```
        const x = 10;
        x = 20; // Error: Assignment to constant variable

        const obj = { name: "Example" };
        obj.name = "Updated Example"; // Allowed: Modifying a property of
the object
        console.log(obj.name); // Output: Updated Example

        const arr = [1, 2, 3];
        arr.push(4); // Allowed: Modifying the array
        console.log(arr); // Output: [1, 2, 3, 4]

        // arr = [5, 6, 7]; // Error: Assignment to constant variable (the
array itself)
```

Summary Table:

Feature	`var`	`let`	`const`
Scope	Function/Global	Block	Block
Hoisting	Hoisted and initialized to `undefined`	Hoisted, Temporal Dead Zone	Hoisted, Temporal Dead Zone
Redeclaration	Allowed	Not allowed	Not allowed
Reassignment	Allowed	Allowed	Not allowed (mostly)

Best Practices:

- Use `let` and `const` by default. They provide better scope control and help prevent common JavaScript errors.
- Use `const` when you know a variable's value will not change. This helps make your code more readable and less prone to bugs. If the variable will hold an object or

array and you intend to modify its properties or elements, it's still appropriate to use `const` as long as you don't reassign the variable itself to a different object or array.

- Avoid using `var` in modern JavaScript unless you have a specific reason to use function scope or are working with older codebases. Block scope is almost always preferable.