Let, const and var

1. Introduction: Understanding Variable Declarations

- The video teaches you how JavaScript handles variables using `var`, `let`, and `const`.
- These keywords decide *where* and *how* a variable can be used, as well as whether it can be changed later.

---

2. `var` – The Old Way

- **Scope**: Function-scoped or globally scoped if used outside a function.
- **Hoisting**: Variables declared with `var` are hoisted to the top of their scope, meaning they exist earlier in the code (initialized as `undefined`).
- **Reassignable & Redeclarable**:

```
var x = 1;
var x = 2; // no error
x = 3;     // works fine
```

- **Downside**: Can lead to confusing bugs when used inside loops or functions without clear scoping.

---

3. `let` – Block Scoped, Better Control

- **Scope**: Block-scoped (`{...}` only), preventing accidental use outside its intended area.

```
{
  let a = 10;
}
console.log(a); // Error: a is not defined
```

- **Hoisting**: Technically hoisted but not accessible until after its declaration ("temporal dead zone").
- **Reassignable, But Not Redeclarable**:

```
let b = 5;
b = 6;       // OK
let b = 7;    // Error: Identifier 'b' has already been declared
```

- **Advantage**: Keeps code cleaner and safer, especially inside loops and nested blocks.

---

4. `const` – Immutable References

- **Scope & Hoisting**: Same as `let` (block-scoped + temporal dead zone).
- **Cannot be Reassigned or Redeclared**:

```
const PI = 3.14;
PI = 3.15;      // Error: Assignment to constant variable.
const PI = 3.14; // Error: Identifier 'PI' has already been declared
```

- **Mutable Contents**: If the value is an object or array, the contents can still change:

```
const arr = [1,2,3];
arr.push(4); // OK
arr = [];    // Error: cannot reassign
```

- **Best Practice**: Use `const` for anything that shouldn't be reassigned, which improves code clarity and intent.

---

5. Illustrative Code Examples

- The presenter runs small interactive examples to show how each behaves in real-time:
  - Revising a `for` loop using `var` vs `let`

- Attempting to change values declared with `const`
- Observing what breaks when scoping rules aren't followed

---

6. When to Use Each

- 💡 `var`: Avoid in modern JavaScript—use only if maintaining legacy code.
- ✅ `let`: Use for values that need to change within a block (e.g., loop counters).
- 🔒 `const`: Prefer for everything else—keeps data integrity and intention visible.

---

7. Tips & Best Practices

- Default to `const` unless you need to change the value, then use `let`.
- Never redeclare variables unintentionally.
- Stick to one style (const > let > avoid var) consistently to make code easier to read and debug.
- Exploit block scoping to avoid global namespace pollution.

📌 Key Points (Quick Notes for Reviewing)

```
- var: function/global scoped, hoisted, re-declarable — avoid for new code.
- let: block-scoped, TDZ, reassignable, safer in loops/blocks.
- const: block-scoped, TDZ, no reassignment, but mutable object contents.
- Favor const → let → (var only for legacy).
- Use block scoping to avoid bugs.
```

📝 Copy-Paste Friendly Revision Notes

```
## JS Variable Declarations

### `var`
- Function or global scope
- Hoisted (initialized undefined)
```

- Redeclarable and reassignable
- ⚠️ Use only in legacy code

`let`
- Block scoped
- Temporal Dead Zone (TDZ): cannot use before declaration
- Reassignable, but no redeclaration
- 👍 Good for loops and block-specific variables

`const`
- Block scoped + TDZ
- No reassign or redeclare
- Objects/arrays can mutate
- 🔒 Default choice for variables

Best Practices
- Use **const** by default
- Switch to **let** for mutable values
- Avoid **var**
- Leverage block scoping for safer, cleaner code