In TypeScript, declaring variables using `let` is generally considered better practice than using `var`. This preference stems from the differences in their scoping behavior and how they handle variable hoisting and redeclaration.

Reasons why `let` is preferred over `var`:

- **Block Scoping:**

  `let` declares variables with block scope, meaning the variable is only accessible within the code block (e.g., `if` statements, `for` loops, or function bodies) where it is declared. This helps prevent unintended variable access and modification outside of their intended scope. In contrast, `var` has function scope or global scope, which can lead to unexpected behavior in larger codebases.

- **No Redeclaration in the Same Scope:**

  `let` prevents redeclaring a variable with the same name within the same block scope, which helps catch potential naming conflicts and errors during development. `var`, however, allows redeclaration, which can lead to accidental overwriting of variables.

- **Reduced Hoisting Issues:**
  While both `var` and `let` declarations are hoisted (moved to the top of their scope during compilation), `let` variables are not initialized until their declaration is encountered during execution. Accessing a `let` variable before its declaration results in a `ReferenceError`, promoting clearer and more predictable code. `var` variables, on the other hand, are initialized to `undefined` when hoisted, which can mask bugs.

When `var` might still be encountered:

- **Legacy Code:**

  You might encounter `var` in older JavaScript or TypeScript codebases that were written before `let` and `const` were introduced (ES2015/ES6).

- **Specific Scenarios:**
  In very rare and specific cases, where function-scoped behavior is explicitly desired and understood, `var` might be used, though this is uncommon in modern TypeScript development.

Conclusion:

For modern TypeScript development, `let` (and `const` for variables whose values should not be reassigned) is the recommended choice for variable declarations due to its block-scoping, prevention of redeclaration, and clearer hoisting behavior, which leads to more robust and maintainable code.