Case 1: Inside a **Composable**

```
@Composable
fun MyScreen() {
    var points by mutableIntStateOf(0)

    Button(onClick = { points++ }) {
        Text("Points: $points")
    }
}
```

✅ This **will update the UI** when you press the button.
Because `mutableIntStateOf` is observable state, Compose knows it should recompose when the value changes.

❌ But… the problem:

- Every time `MyScreen` recomposes, the line `var points by mutableIntStateOf(0)` runs again.
- That **resets points back to 0**.
- So the counter never really works as expected.

That's why we need `remember`.

Case 2: With `remember` inside Composable

```
@Composable
fun MyScreen() {
    var points by remember { mutableIntStateOf(0) }

    Button(onClick = { points++ }) {
        Text("Points: $points")
    }
}
```

✅ Now `points` survives recompositions.

- First time → Compose runs the initializer (`0`).
- Next recompositions → Compose gives back the same state object instead of resetting.

Case 3: Inside a **ViewModel**

```
class MainViewModel : ViewModel() {
    var points by mutableIntStateOf(0)
        private set

    fun increasePoints() { points++ }
}
```

✅ Here, no `remember` is needed because:

- ViewModel itself survives recomposition and configuration changes.
- Its properties aren't re-initialized on recomposition.

- `mutableStateOf` ensures Compose observes changes and refreshes UI.

🎯 Final Answer

- **Composable without remember + mutableStateOf** → UI updates, but state resets on recomposition → not good.
- **Composable with remember + mutableStateOf** → UI updates + state survives recomposition → correct way.
- **ViewModel with mutableStateOf** → UI updates + state survives recomposition + config changes → best for app-level state.
