Singleton Design Pattern with volatile and synchronized

1■■ Why volatile is Needed:

When creating a singleton instance with double-checked locking, object creation is not an atomic operation. The JVM can reorder instructions such that another thread sees a reference to the singleton instance before the object is fully initialized. Declaring the instance as 'volatile' prevents this issue by ensuring memory visibility and ordering guarantees.

2■■ Steps Without volatile:
1. Allocate memory for Singleton object.
2. Initialize the object (constructor runs).
3. Assign reference to instance.

Because of instruction reordering, steps 2 and 3 may switch, causing other threads to see a partially constructed object.

3■■ Steps With volatile:
1. Allocate memory.
2. Initialize object.
3. Assign reference to instance.

volatile ensures proper order and visibility across threads.

4■■ Double-Checked Locking Code:

```
public class Singleton {
private static volatile Singleton instance;

private Singleton() {}

public static Singleton getInstance() {
if (instance == null) {
synchronized (Singleton.class) {
if (instance == null) {
instance = new Singleton();
}
}
}
return instance;
}
}
```

5■■ Key Points:
- Lazy initialization (object created only when needed)
- Thread safety with synchronized block
- Performance gain by avoiding unnecessary locking
- volatile ensures no partially initialized object is visible to other threads

6■■ Alternative Approaches:
- Eager initialization
- Bill Pugh Singleton (static inner class for thread-safe lazy initialization without synchronization overhead).