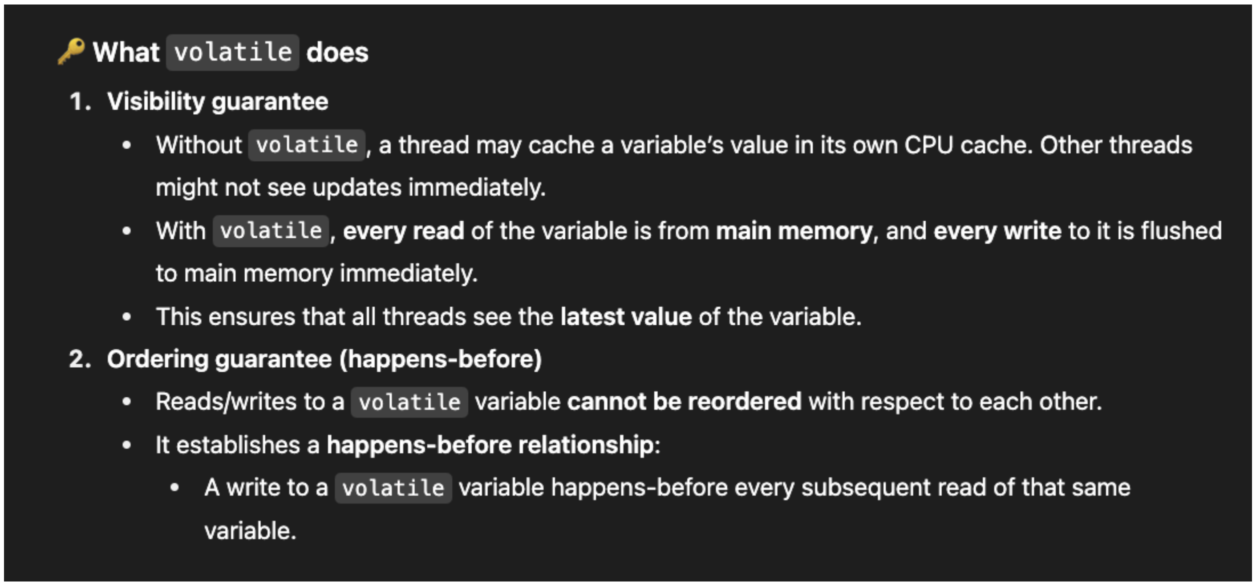
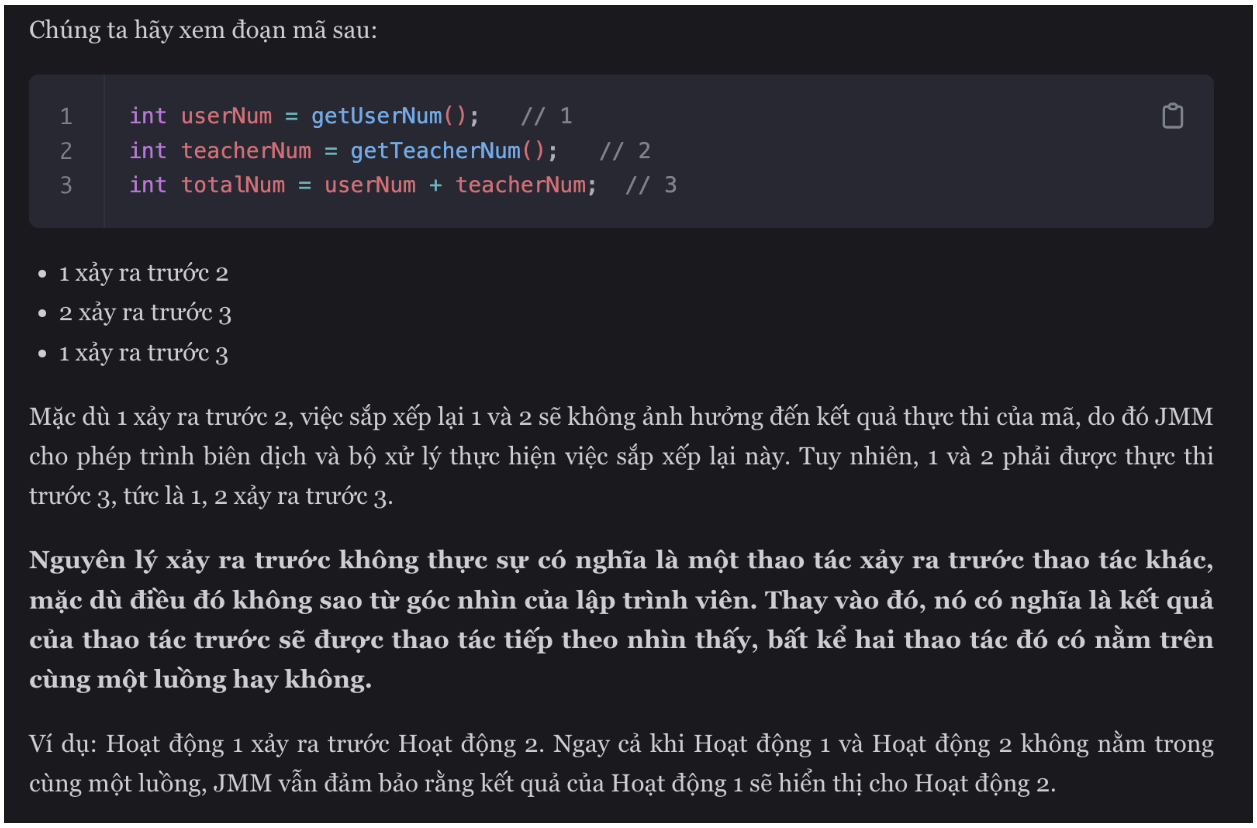


🔑 **What `volatile` does**

1. **Visibility guarantee**
   - Without `volatile`, a thread may cache a variable's value in its own CPU cache. Other threads might not see updates immediately.
   - With `volatile`, **every read** of the variable is from **main memory**, and **every write** to it is flushed to main memory immediately.
   - This ensures that all threads see the **latest value** of the variable.
2. **Ordering guarantee (happens-before)**
   - Reads/writes to a `volatile` variable **cannot be reordered** with respect to each other.
   - It establishes a **happens-before relationship**:
     - A write to a `volatile` variable happens-before every subsequent read of that same variable.

1. Reorder the code (make it parallel and faster)



Chúng ta hãy xem đoạn mã sau:

```java
int userNum = getUserNum();   // 1
int teacherNum = getTeacherNum();   // 2
int totalNum = userNum + teacherNum;  // 3
```

- 1 xảy ra trước 2
- 2 xảy ra trước 3
- 1 xảy ra trước 3

Mặc dù 1 xảy ra trước 2, việc sắp xếp lại 1 và 2 sẽ không ảnh hưởng đến kết quả thực thi của mã, do đó JMM cho phép trình biên dịch và bộ xử lý thực hiện việc sắp xếp lại này. Tuy nhiên, 1 và 2 phải được thực thi trước 3, tức là 1, 2 xảy ra trước 3.

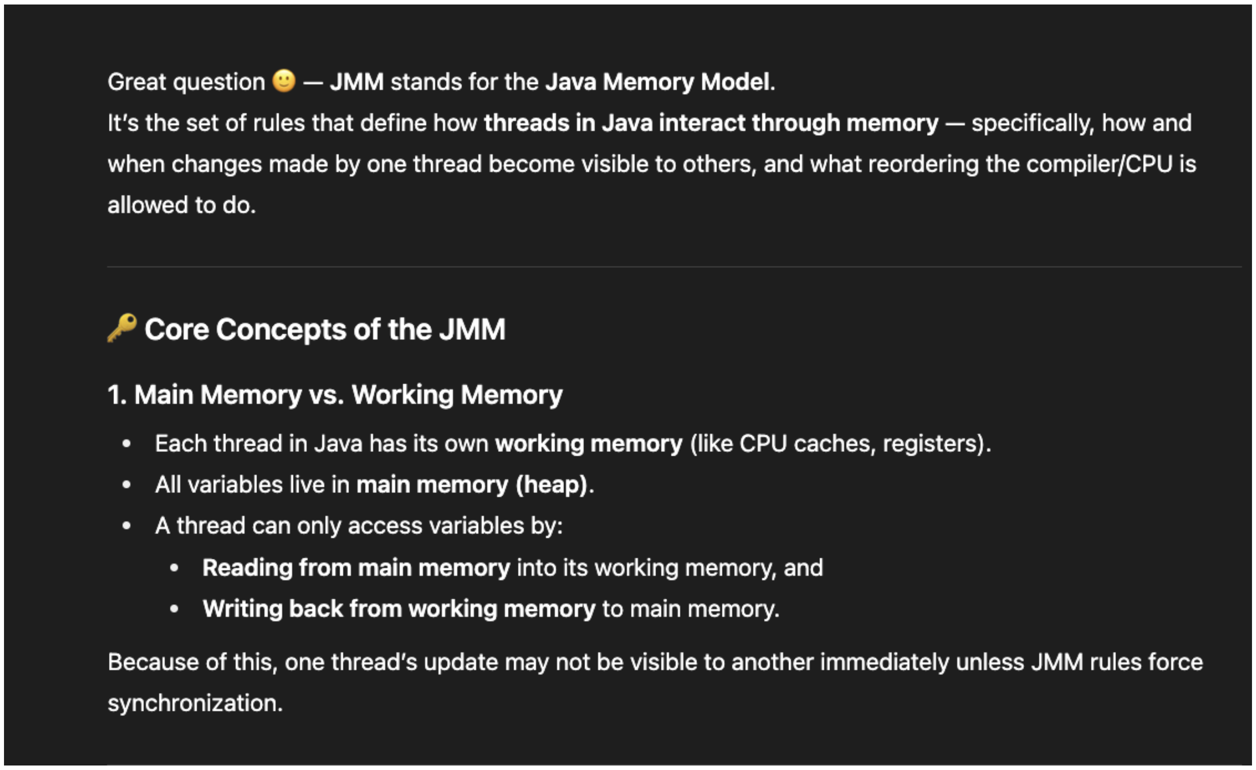
Nguyên lý xảy ra trước không thực sự có nghĩa là một thao tác xảy ra trước thao tác khác, mặc dù điều đó không sao từ góc nhìn của lập trình viên. Thay vào đó, nó có nghĩa là kết quả của thao tác trước sẽ được thao tác tiếp theo nhìn thấy, bất kể hai thao tác đó có nằm trên cùng một luồng hay không.

Ví dụ: Hoạt động 1 xảy ra trước Hoạt động 2. Ngay cả khi Hoạt động 1 và Hoạt động 2 không nằm trong cùng một luồng, JMM vẫn đảm bảo rằng kết quả của Hoạt động 1 sẽ hiển thị cho Hoạt động 2.

2. JMM

The data sync only when the thread writing from its own Thread Memory -> Main Memory

=> Maybe it cause the not visibility at the same time.



Great question 🙂 — **JMM** stands for the **Java Memory Model.**
It's the set of rules that define how **threads in Java interact through memory** — specifically, how and when changes made by one thread become visible to others, and what reordering the compiler/CPU is allowed to do.

🔑 Core Concepts of the JMM

1. Main Memory vs. Working Memory

- Each thread in Java has its own **working memory** (like CPU caches, registers).
- All variables live in **main memory (heap)**.
- A thread can only access variables by:
  - **Reading from main memory** into its working memory, and
  - **Writing back from working memory** to main memory.

Because of this, one thread's update may not be visible to another immediately unless JMM rules force synchronization.

3. Happens-Before Relationship

The JMM defines a set of **happens-before rules** that ensure visibility and ordering. If one action *happens-before* another, then:

- The first action's result is visible to the second, and
- Reordering between them is not allowed.

4. Atomicity, Visibility, Ordering

- Atomicity → indivisible operations (e.g., reads/writes to `int`/`boolean` are atomic, but `long`/`double` weren't atomic before Java 5).

- Visibility → when one thread updates a value, other threads see it => **In Main Memory**

- Ordering → the order of execution seen by one thread matches the intended order => **Do not reorder the code.**

---

5. Why `long` and `double` were not atomic (before Java 5) => after Java 5 it is atomic

- On **32-bit JVMs**, a `long` (64-bit) or `double` (64-bit) doesn't fit into a single machine word.

- So, a read/write might need **two separate 32-bit operations**.

6. How `synchronized` ensures atomicity

1. Mutual exclusion (atomicity)
    - Only **one thread at a time** can execute inside a `synchronized` block on the same object.
    - That prevents race conditions.
2. Visibility
    - When a thread **exits** a `synchronized` block, it **flushes changes** from its working memory to main memory.
    - When another thread **enters**, it **invalidates its cache** and reads the latest values from main memory.

=> The data is always up to date in main memory.

7. How `AtomicInteger` ensures atomicity (CAS giống khoá lạc quan)

`AtomicInteger` (and other classes in `java.util.concurrent.atomic`) use **CAS (Compare-And-Swap)** — a low-level **lock-free** CPU instruction.

How CAS works:

1. Read the current value from memory.
2. Compare it with the expected value.
3. If it’s the same, swap it with the new value (in one atomic CPU instruction).
4. If another thread already changed it, retry (loop until success).

8. How reordering improves performance

1. Instruction-level parallelism (ILP)

Modern CPUs have **multiple execution units** (ALU, FPU, load/store units).
 If instructions are independent, the CPU can run them **in parallel**.

Example:

```
int a = 1;   // store to register R1

int b = 2;   // store to register R2

int c = a + b; // needs R1, R2
```

- The assignment `a = 1` and `b = 2` don’t depend on each other.
- CPU can reorder or **issue both at once** to different execution units.
- Only `c = a + b` must wait until both are ready.

This is called **out-of-order execution** — CPU schedules instructions to keep pipelines full.