Volatile

Explanation

The volatile concept is specific to Java. Its easier to understand volatile, if you understand the problem it solves.

If you have a variable say a counter that is being worked on by a thread, it is possible the thread keeps a copy of the counter variable in the CPU cache and manipulates it rather than writing to the main memory. The JVM will decide when to update the main memory with the value of the counter, even though other threads may read the value of the counter from the main memory and may end up reading a stale value.

If a variable is declared volatile then whenever a thread writes or reads to the volatile variable, the read and write always happen in the main memory. As a further guarantee, all the variables that are visible to the writing thread also get written-out to the main memory alongside the volatile variable. Similarly, all the variables visible to the reading thread alongside the volatile variable will have the latest values visible to the reading thread.

Example

Consider the program below:

```
public class VolatileExample {
```

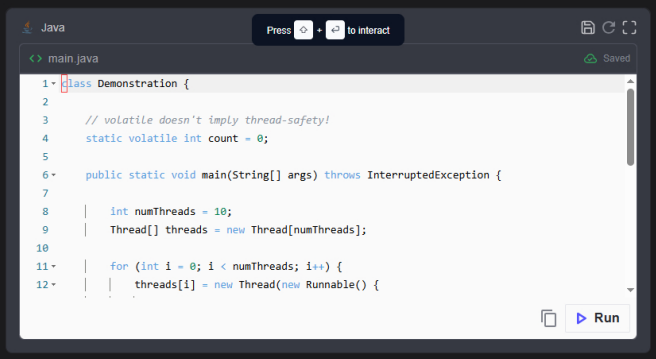
the variables that are visible to the writing thread also get written-out to the main memory alongside the volatile variable. Similarly, all the variables visible to the reading thread alongside the volatile variable will have the latest values visible to the reading thread.

Example

Consider the program below:

```
public class VolatileExample {

    boolean flag = false;

    void threadA() {

        while (!flag) {

            // ... Do something useful
        }

    }

    void threadB() {
        flag = true;
    }
}
```

In the above program, we would expect that `threadA` would exit the `while` loop once `threadB` sets the variable `flag` to true but `threadA` may unfortunately find itself spinning forever if it has cached the variable `flag`'s value. In this scenario, marking `flag` as `volatile` will fix the problem. Note that `volatile` presents a consistent view of the memory to all the threads. However, remember **that `volatile` doesn't imply or mean thread-safety**. Consider the program below where we declare the variable `count` `volatile` and several threads increment the variable a 1000 times each. If you run the program several times you'll see `count` summing up to values other than the expected 10,000.



When is `volatile` thread-safe

Volatile comes into play because of multiples levels of memory in hardware architecture required for performance enhancements. If there's a single thread that writes to the volatile variable and other threads only read the volatile variable then just using volatile is enough

variable and other threads only read the volatile variable then just using volatile is enough, however, if there's a possibility of multiple threads writing to the volatile variable then "synchronized" would be required to ensure atomic writes to the variable.

← Back lesson
Interrupting Threads

Completed Next →
Reentrant Locks & Condition Variables