Difference Between Atomic, Volatile and Synchronized in Java

Last Updated : 17 Nov, 2021

Synchronized is the modifier applicable only for methods and blocks but not for the variables and for classes. There may be a chance of data inconsistency problem to overcome this problem we should go for a synchronized keyword when multiple threads are trying to operate simultaneously on the same java object. If a method or block declares as synchronized then at a time only one thread at a time is allowed to execute that method or block on the given object so that the data inconsistence problem will be resolved. The main advantage of synchronized keywords is we can resolve data inconsistence problems but the main disadvantage of this keyword is it increases the waiting time of the thread and creates performance problems. Hence, It is not recommended using, the synchronized keyword when there is no specific requirement. Every object in java has a unique lock. The lock concept will come into the picture when we are using a synchronized keyword.

The remaining threads are not allowed to execute any synchronized method simultaneously on the same object when a thread executing a synchronized method on the given object. But remaining threads are allowed to execute the non-synchronized method simultaneously.

Volatile Modifier:

If a value of a variable keeps on changing by multiple threads then there may be a chance of a data inconsistency problem. It is a modifier applicable only for variables, and we can't apply it anywhere else. We can solve this problem by using a volatile modifier. If a variable is declared as volatile as for every thread **JVM** will create a separate local copy. Every modification performed by the thread will take place in the local copy so that there is no effect on the remaining threads. Overcoming the data inconsistency problem is the advantage and the volatile keyword is creating and maintaining a separate copy for every thread increases the complexity of the programming and creates performance problem is a disadvantage. Hence, if there are no specific requirements it is never recommended to use volatile keywords.

Atomic Modifier:

If a value of a variable keeps on changing by multiple threads then there may be a chance of a data inconsistency problem. We can solve this problem by using an atomic variable. Data inconsistency problem can be solved when objects of these classes represent the atomic variable of int, long, boolean, and object reference respectively.

Example:

In the below example every thread increments the count variable 5 times. So after the execution of two threads, the finish count value should be 10.

```
// import required packages
import java.io.*;
import java.util.*;

// creating a thread by extending a thread class
class myThread extends Thread {

    // declaring a count variable
    private int count;

    public void run()
    {
        // calculating the count
        for (int i = 1; i <= 5; i++) {

            try {
                Thread.sleep(i * 100);
                count++;
            }
            catch (InterruptedException
                       e) { // throwing an exception
                System.out.println(e);
            }
        }
    }
    // returning the count value
    public int getCount() { return this.count; }
}
// driver class
public class GFG {

    // main method
    public static void main(String[] args)
        throws InterruptedException
    {

        // creating an thread object
        myThread t = new myThread();
        Thread t1 = new Thread(t, "t1");

        // starting thread t1
        t1.start();
        Thread t2 = new Thread(t, "t2");
```

```
        // starting thread t2
        t2.start();

        // calling join method on thread t1
        t1.join();

        // calling join method on thread t1
        t2.join();

        // displaying the count
        System.out.println("count=" + t.getCount());
    }
}
```

Output

```
count=10
```

If we run the above program, we will notice that the count value varies between 6,7,8.9 The reason is that count++ is not an atomic operation. So by the time one thread read itâ€™s value and increment it by one, another thread has read the older value leading to the wrong result. To solve this issue, we will have to make sure that increment operation on count is atomic.

Below program will always output count value as 8 because AtomicInteger method incrementAndGet() atomically increments the current value by one.

```
// import required packages
import java.io.*;
import java.util.*;
import java.util.concurrent.atomic.AtomicInteger;

// creating a thread by extending a thread class
class myThread extends Thread {

    // declaring an atomic variable
    private AtomicInteger count = new AtomicInteger();
```

```
    public void run()
    {
        // calculating the count
        for (int i = 1; i <= 5; i++) {
            try {

                // putting  thread on sleep
                Thread.sleep(i * 100);

                // calling incrementAndGet() method
                // on count variable
                count.incrementAndGet();
            }
            catch (InterruptedException e) {

                // throwing exception
                System.out.println(e);
            }
        }
    }
    // returning the count value
    public AtomicInteger getCount() { return count; }
}
// driver class
public class GFG {

    // main method
    public static void main(String[] args)
        throws InterruptedException
    {
        // creating an thread object
        myThread t = new myThread();

        Thread t1 = new Thread(t, "t1");

        // starting thread t1
        t1.start();

        Thread t2 = new Thread(t, "t2");

        // starting thread t2
        t2.start();

        // calling join method on thread t1
        t1.join();

        // calling join method on thread t1
        t2.join();

        // displaying the count
        System.out.println("count=" + t.getCount());
    }
}
```

Output

```
count=10
```

Synchronized	Volatile	Atomic
1.It is applicable to only blocks or methods.	1.It is applicable to variables only.	1.It is also applicable to variables only.
2. Synchronized modifier is used to implement a lock-based concurrent algorithm, and i.e it suffers from the limitation of locking.	2.Whereas Volatile gives the power to implement a non-blocking algorithm that is more scalable.	2.Atomic also gives the power to implement the non-blocking algorithm.
3.Performance is relatively low compare to volatile and atomic keywords because of the acquisition and release of the lock.	3.Performance is relatively high compare to synchronized keyword.	3.Performance is relatively high compare to both volatile and synchronized keyword.
4.Because of its locking nature it is not immune to concurrency hazards such as deadlock and livelock.	4.Because of its non-locking nature it is immune to concurrency hazards such as deadlock and livelock.	4.Because of its non-locking nature it is immune to concurrency hazards such as deadlock and livelock.