Concurrency

To modify the value of a variable by different threads

Why to use Volatile

- to make classes thread safe.
- multiple threads can use a method and instance of the classes at the same time without any problem
- Can be used either with primitive type or objects.
- Does not cache the value of the variable and always read the variable from the main memory
- Cannot be used with classes or methods.

Why to use Volatile

- Used with variables only.
- It also guarantees visibility and ordering.
- It prevents the compiler from the reordering of code.
- The contents of the particular device register could change at any time, so you need the volatile keyword to ensure that such accesses are not optimized away by the compiler.

Problem Statement

```
class Test
{
    static int var=5;
}
```

- Two threads are working on the same class. Both threads run on different processors where each thread has its local copy of var. If any thread modifies its value, the change will not reflect in the original one in the main memory. It leads to data inconsistency because the other thread is not aware of the modified value

Problem Statement Solution

```
class Test
{
static volatile int var =5;
}
```

- static variables are class members that are shared among all objects. There is only one copy in the main memory.
- The value of a volatile variable will never be stored in the cache.
- All read and write will be done from and to the main memory.

When to Use

- If you want to read and write long and double variable automatically.
- As an alternative way of achieving synchronization in Java.
- All reader threads will see the updated value of the volatile variable after completing the write operation. If you are not using the volatile keyword, different reader thread may see different values.
- Used to inform the compiler that multiple threads will access a particular statement. It prevents the compiler from doing any reordering or any optimization.
- If you do not use volatile variable compiler can reorder the code, free to write in cache value of volatile variable instead of reading from the main memory.

Synchronization vs Volatile

Volatile	Synchronization
Volatile keyword is a field modifier	Synchronized keyword modifies code blocks and methods.
The thread cannot be blocked for waiting in case of volatile.	Threads can be blocked for waiting in case of synchronized.
It improves thread performance.	Synchronized methods degrade the thread performance.
It synchronizes the value of one variable at a time between thread memory and main memory.	It synchronizes the value of all variables between thread memory and main memory.
Volatile fields are not subject to compiler optimization.	Synchronize is subject to compiler optimization.

Example of Volatile Keyword

```
public class VolatileData
{
private volatile int counter = 0;
public int getCounter()
{
return counter;
}
public void increaseCounter()
{
++counter;
 //increases the value of counter by 1
}
}
```

```
public class VolatileThread extends Thread
{
private final VolatileData data;
public VolatileThread(VolatileData data)
{
this.data = data;
}
@Override
public void run()
{
int oldValue = data.getCounter();
System.out.println("[Thread " + Thread.currentThread().getId() + "]: Old
value = " + oldValue);
data.increaseCounter();
int newValue = data.getCounter();
System.out.println("[Thread " + Thread.currentThread().getId() + "]: New
value = " + newValue);
}
}
```

Example of Volatile Keyword

```
public class VolatileMain
{

private final static int noOfThreads = 2;
public static void main(String[] args) throws InterruptedException
{
VolatileData volatileData = new VolatileData();     //object of VolatileData class
Thread[] threads = new Thread[noOfThreads];     //creating Thread array
for(int i = 0; i < noOfThreads; ++i)
threads[i] = new VolatileThread(volatileData);
for(int i = 0; i < noOfThreads; ++i)
threads[i].start();                 //starts all reader threads
for(int i = 0; i < noOfThreads; ++i)
threads[i].join();                  //wait for all threads
}
}
```

Deadlock in Java

- Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.
- A deadlock may also include more than two threads. The reason is that it can be difficult to detect a deadlock. Here is an example in which four threads have deadlocked:

Deadlock Example

- Thread 1 locks A, waits for B
- Thread 2 locks B, waits for C
- Thread 3 locks C, waits for D
- Thread 4 locks D, waits for A
- Thread 1 waits for thread 2, thread 2 waits for thread 3, thread 3 waits for thread 4, and thread 4 waits for thread 1.

```
public class TestDeadlockExample1 {
  public static void main(String[] args) {
    final String resource1 = "ratan jaiswal";
    final String resource2 = "vimal jaiswal";
    // t1 tries to lock resource1 then resource2
    Thread t1 = new Thread() {
      public void run() {
          synchronized (resource1) {
           System.out.println("Thread 1: locked resource 1");

           try { Thread.sleep(100);} catch (Exception e) {}

           synchronized (resource2) {
            System.out.println("Thread 1: locked resource 2");
           }
          }
      }
    };

    // t2 tries to lock resource2 then resource1
    Thread t2 = new Thread() {
      public void run() {
        synchronized (resource2) {
          System.out.println("Thread 2: locked resource 2");

          try { Thread.sleep(100);} catch (Exception e) {}

          synchronized (resource1) {
            System.out.println("Thread 2: locked resource 1");
          }
        }
      }
    };

    t1.start();
    t2.start();
  }
}
```

Deadlock Prevention

- A solution for a problem is found at its roots. In deadlock it is the pattern of accessing the resources A and B, is the main issue. To solve the issue we will have to simply re-order the statements where the code is accessing shared resources.
- Avoid Nested Locks: We must avoid giving locks to multiple threads, this is the main reason for a deadlock condition. It normally happens when you give locks to multiple threads.
- Avoid Unnecessary Locks: The locks should be given to the important threads. Giving locks to the unnecessary threads that cause the deadlock condition.
- Using Thread Join: A deadlock usually happens when one thread is waiting for the other to finish. In this case, we can use join with a maximum time that a thread will take.

```
public class DeadlockSolved {
    public static void main(String ar[]) {
        DeadlockSolved test = new DeadlockSolved();

        final resource1 a = test.new resource1();
        final resource2 b = test.new resource2();

      // Thread-1
Runnable b1 = new Runnable() {
    public void run() {
        synchronized (b) {
            try {
                /* Adding delay so that both threads can start trying to lock resources */
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            // Thread-1 have resource1 but need resource2 also
            synchronized (a) {
                System.out.println("In block 1");
            }
        }
    }
};

// Thread-2
Runnable b2 = new Runnable() {
    public void run() {
        synchronized (b) {
            // Thread-2 have resource2 but need resource1 also
            synchronized (a) {
                System.out.println("In block 2");
            }
        }
    }
};
        new Thread(b1).start();
        new Thread(b2).start();
    }

    // resource1
    private class resource1 {
        private int i = 10;

        public int getI() {
            return i;
        }

        public void setI(int i) {
            this.i = i;
        }
    }

    // resource2
    private class resource2 {
        private int i = 20;

        public int getI() {
            return i;
        }

        public void setI(int i) {
            this.i = i;
        }
    }
}
```

Need of Concurrent Collections

- Most of the Collections classes objects (like ArrayList, LinkedList, HashMap etc) are non-synchronized in nature i.e. multiple threads can perform on a object at a time simultaneously. Therefore objects are not thread-safe.
- Very few Classes objects (like Vector, Stack, HashTable) are synchronized in nature i.e. at a time only one thread can perform on an Object. But here the problem is performance is low because at a time single thread execute an object and rest thread has to wait.
- The main problem is when one thread is iterating an Collections object then if another thread cant modify the content of the object. If another thread try to modify the content of object then we will get RuntimeException saying ConcurrentModificationException.
- Because of the above reason Collections classes is not suitable or we can say that good choice for Multi-threaded applications.

```
class ConcurrentDemo extends Thread {
static ArrayList l = new ArrayList();
public void run()
{
try {
Thread.sleep(2000);
}
catch (InterruptedException e) {
System.out.println("Child Thread" +
" going to add element");
}

// Child thread trying to add new
// element in the Collection object
l.add("D");
}

public static void main(String[] args)
throws InterruptedException
{
l.add("A");
l.add("B");
l.add("c");

// We create a child thread that is
// going to modify ArrayList l.
ConcurrentDemo t = new ConcurrentDemo();
t.start();

// Now we iterate through the ArrayList
// and get exception.
Iterator itr = l.iterator();
while (itr.hasNext()) {
String s = (String)itr.next();
System.out.println(s);
Thread.sleep(6000);
}
System.out.println(l);
}
}
```

Need of Concurrent Collections

- BlockingQueue defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue.
- ConcurrentMap is a subinterface of java.util.Map that defines useful atomic operations. These operations remove or replace a key-value pair only if the key is present, or add a key-value pair only if the key is absent. Making these operations atomic helps avoid synchronization. The standard general-purpose implementation of ConcurrentMap is ConcurrentHashMap, which is a concurrent analog of HashMap.
- ConcurrentNavigableMap is a subinterface of ConcurrentMap that supports approximate matches. The standard general-purpose implementation of ConcurrentNavigableMap is ConcurrentSkipListMap, which is a concurrent analog of TreeMap.

Features of Stream In Java

- A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
- Streams don’t change the original data structure, they only provide the result as per the pipelined methods.
- Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

Stream Intermediate Operation

- map: The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.

```
List number = Arrays.asList(2,3,4,5);
List square = number.stream().map(x->x*x).collect(Collectors.toList());
```

- filter: The filter method is used to select elements as per the Predicate passed as argument.

```
List names = Arrays.asList("Reflection","Collection","Stream");
List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());
```

- sorted: The sorted method is used to sort the stream.

```
List names = Arrays.asList("Reflection","Collection","Stream");
List result = names.stream().sorted().collect(Collectors.toList());
```

Stream Terminal Operation

- collect: The collect method is used to return the result of the intermediate operations performed on the stream.

```
List number = Arrays.asList(2,3,4,5,3);
Set square = number.stream().map(x->x*x).collect(Collectors.toSet());
```

- forEach: The forEach method is used to iterate through every element of the stream.

```
List number = Arrays.asList(2,3,4,5);
number.stream().map(x->x*x).forEach(y->System.out.println(y));
```

- reduce: The reduce method is used to reduce the elements of a stream to a single value. The reduce method takes a BinaryOperator as a parameter.

```
List number = Arrays.asList(2,3,4,5);
int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);
```

```
class Demo
{
public static void main(String args[])
{

// create a list of integers
List<Integer> number = Arrays.asList(2,3,4,5);

// demonstration of map method
List<Integer> square = number.stream().map(x -> x*x).
collect(Collectors.toList());
System.out.println(square);

// create a list of String
List<String> names =
Arrays.asList("Reflection","Collection","Stream");

// demonstration of filter method
List<String> result = names.stream().filter(s->s.startsWith("S")).
collect(Collectors.toList());
System.out.println(result);

// demonstration of sorted method
List<String> show =
names.stream().sorted().collect(Collectors.toList());
System.out.println(show);

// create a list of integers
List<Integer> numbers = Arrays.asList(2,3,4,5,2);

// collect method returns a set
Set<Integer> squareSet =
numbers.stream().map(x->x*x).collect(Collectors.toSet());
System.out.println(squareSet);

// demonstration of forEach method
number.stream().map(x->x*x).forEach(y->System.out.println(y));

// demonstration of reduce method
int even =
number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);

System.out.println(even);
}
}
```

Points to be Noted

- A stream consists of source followed by zero or more intermediate methods combined together (pipelined) and a terminal method to process the objects obtained from the source as per the methods described.
- Stream is used to compute elements as per the pipelined methods without altering the original value of the object.

Date Time API

- Introduced in JDK 8, are a set of packages that model the most important aspects of date and time. The core classes in the **java.time** package use the calendar system defined in ISO-8601 (based on the Gregorian calendar system) as the default calendar.
- Java 8 introduces a new date-time API under the package **java.time**
- Local − Simplified date-time API with no complexity of timezone handling.
- Zoned − Specialized date-time API to deal with various timezones.

Why Date Time API

- Not thread safe − java.util.Date is not thread safe, thus developers have to deal with concurrency issue while using date. The new date-time API is immutable and does not have setter methods.
- Poor design − Default Date starts from 1900, month starts from 1, and day starts from 0, so no uniformity. The old API had less direct methods for date operations. The new API provides numerous utility methods for such operations.
- Difficult time zone handling − Developers had to write a lot of code to deal with timezone issues. The new API has been developed keeping domain-specific design in mind.

```
public class Java8Tester {

   public static void main(String args[]) {
      Java8Tester java8tester = new Java8Tester();
      java8tester.testLocalDateTime();
   }

   public void testLocalDateTime() {
      // Get the current date and time
      LocalDateTime currentTime = LocalDateTime.now();
      System.out.println("Current DateTime: " + currentTime);

      LocalDate date1 = currentTime.toLocalDate();
      System.out.println("date1: " + date1);

      Month month = currentTime.getMonth();
      int day = currentTime.getDayOfMonth();
      int seconds = currentTime.getSecond();

      System.out.println("Month: " + month +"day: " + day +"seconds: " + seconds);

      LocalDateTime date2 = currentTime.withDayOfMonth(10).withYear(2012);
      System.out.println("date2: " + date2);

      //12 december 2014
      LocalDate date3 = LocalDate.of(2014, Month.DECEMBER, 12);
      System.out.println("date3: " + date3);

      //22 hour 15 minutes
      LocalTime date4 = LocalTime.of(22, 15);
      System.out.println("date4: " + date4);

      //parse a string
      LocalTime date5 = LocalTime.parse("20:15:30");
      System.out.println("date5: " + date5);
   }
}
```

Zoned Date Time API

```
public class Java8Tester {

   public static void main(String args[]) {
      Java8Tester java8tester = new Java8Tester();
      java8tester.testZonedDateTime();
   }

   public void testZonedDateTime() {
      // Get the current date and time
      ZonedDateTime date1 = ZonedDateTime.parse("2007-12-03T10:15:30+05:30[Asia/Kolkata]");
      System.out.println("date1: " + date1);

      ZoneId id = ZoneId.of("Europe/Paris");
      System.out.println("ZoneId: " + id);

      ZoneId currentZone = ZoneId.systemDefault();
      System.out.println("CurrentZone: " + currentZone);
   }
}
```

Chrono Units Enum

- java.time.temporal.ChronoUnit enum is added in Java 8 to replace the integer values used in old API to represent day, month, etc

Chrono Units Enum

```
public class Java8Tester {

   public static void main(String args[]) {
      Java8Tester java8tester = new Java8Tester();
      java8tester.testChromoUnits();
   }

   public void testChromoUnits() {
      //Get the current date
      LocalDate today = LocalDate.now();
      System.out.println("Current date: " + today);

      //add 1 week to the current date
      LocalDate nextWeek = today.plus(1, ChronoUnit.WEEKS);
      System.out.println("Next week: " + nextWeek);

      //add 1 month to the current date
      LocalDate nextMonth = today.plus(1, ChronoUnit.MONTHS);
      System.out.println("Next month: " + nextMonth);

      //add 1 year to the current date
      LocalDate nextYear = today.plus(1, ChronoUnit.YEARS);
      System.out.println("Next year: " + nextYear);

      //add 10 years to the current date
      LocalDate nextDecade = today.plus(1, ChronoUnit.DECADES);
      System.out.println("Date after ten year: " + nextDecade);
   }
}
```

Period and Duration

- With Java 8, two specialized classes are introduced to deal with the time differences.
- Period − It deals with date based amount of time.
- Duration − It deals with time based amount of time.

Period and Duration

```
public class Java8Tester {

   public static void main(String args[]) {
      Java8Tester java8tester = new Java8Tester();
      java8tester.testPeriod();
      java8tester.testDuration();
   }

   public void testPeriod() {
      //Get the current date
      LocalDate date1 = LocalDate.now();
      System.out.println("Current date: " + date1);

      //add 1 month to the current date
      LocalDate date2 = date1.plus(1, ChronoUnit.MONTHS);
      System.out.println("Next month: " + date2);

      Period period = Period.between(date2, date1);
      System.out.println("Period: " + period);
   }

   public void testDuration() {
      LocalTime time1 = LocalTime.now();
      Duration twoHours = Duration.ofHours(2);

      LocalTime time2 = time1.plus(twoHours);
      Duration duration = Duration.between(time1, time2);

      System.out.println("Duration: " + duration);
   }
}
```

Temporal Adjusters

- TemporalAdjuster is used to perform the date mathematics. For example, get the "Second Saturday of the Month" or "Next Tuesday".