Programming Fundamentals 3 **2020**

Instructor: Prof. Walter Binder **TA:** Eduardo Rosales, Filippo Schiavio, Matteo Basso

---

Assignment 2 (6 points) **Due date: 1 November 2020, 08:00 a.m.**

For questions regarding this assignment, please contact Matteo Basso via email: matteo.basso@usi.ch.

This assignment contributes 6% to the overall grade. Please follow strictly the submission instructions written at the end of the assignment.

Environment Setup

The two exercises presented in this assignment are available as maven projects in the attached folders: **/Ex1** and **/Ex2**. Before proceeding, you are advised to properly set the `JAVA_HOME` variable and you must install maven following the official installation guide.

This document contains instructions to use maven from the command line. Alternatively, you can also use the given maven projects from your preferred IDE. The following documentation explain you how to import a project, execute the `main` method, and run tests:

- **IntelliJ IDEA:**
  - Installation
  - Goals configuration (compile, run main, clean, etc.)
  - Testing
- **Eclipse:**
  - The M2Eclipse plugin enables using maven from Eclipse.
  - This tutorial shows you how to import existing maven projects and create new configurations.

Exercise 1 - Counting Palindromic Numbers - Thread Safety (3 points)

Consider **Exercise 2 of Assignment 1: Counting Palindromic Numbers**. The following is a new implementation of class `ThreadPalindromicCounter`, i.e., a Java class that implements a parallel algorithm to count the occurrences of palindromic numbers within an integer array. Method `ThreadPalindromicCounter.count(..)` receives an integer array and returns the number of palindromic numbers within it. Note that in the implementation below, separate threads do not return partial results that need to be collected and aggregated (as it was the case in **Exercise 2 of Assignment 1**). Instead, partial results are accumulated into the shared mutable field `ThreadPalindromicCounter.total`. Please notice that, in the following exercise, we want to avoid only races that involve threads created by class `ThreadPalindromicCounter`. You will not be asked to make the whole class thread-safe. For instance, we do not take into account races that might occur if several threads invoke `ThreadPalindromicCounter.count(..)` concurrently using the same instance.

Note that class `ThreadPalindromicCounter` relies on the inner class `PalindromicCounter`, a `Runnable` implementing tasks that are delegated to count the occurrences of palindromic numbers in distinct segments of the array, updating `ThreadPalindromicCounter.total` upon each occurrence of a palindromic number.

```java
public final class ThreadPalindromicCounter implements Counter {

  private final int numThreads;
  private int total;

  public ThreadPalindromicCounter(int numThreads) {
    this.numThreads = numThreads;
  }

  public int count(int[] nums) throws Exception {
    total = 0;

    List<Thread> threads = new LinkedList<Thread>();
    int size = (int) Math.ceil((double) nums.length / numThreads);
    int threadId = 0;
    for (int low = 0; low < nums.length; low += size) {
      PalindromicCounter counter =
            new PalindromicCounter(nums, low, low + size);
      Thread thread = new Thread(counter);
      thread.start();
      threads.add(thread);
    }

    for (Thread thread : threads) {
      thread.join();
    }

    return total;
  }

  private void incrementCounter() {
    total++;
  }

  private class PalindromicCounter implements Runnable {
    private final int[] nums;
    private final int low, high;

    private PalindromicCounter(int[] nums, int low, int high) {
      this.nums = nums;
      this.low = low;
      this.high = Math.min(high, nums.length);
    }

    @Override
    public void run() {
      for (int i = low; i < high; i++) {
        if (Palindromic.isPalindromic(nums[i])) {
          ThreadPalindromicCounter.this.incrementCounter();
        }
      }
    }
  }
}
```

Class `Main` enables you running the different implementations used in this exercise.

```
public final class Main {

  public static final int LOW = 1;
  public static final int HIGH = 99_999_999;
  private static final int NUM_THREADS = 8;

  public static void main(String[] args) throws Exception  {
    Class<? extends Counter> counterClass = null;

    String counterClassName = args.length > 0
      ? args[0]
      : "ThreadPalindromicCounter";
    try {
      counterClass = Class.forName("assignment2." + counterClassName)
                          .asSubclass(Counter.class);
    } catch (ClassNotFoundException ex) {
      System.err.println(
        "- ERROR: the provided command line parameter is not valid. " +
        "Cannot find a class named \"" + counterClassName + "\""
      );
      return;
    }

    Counter counter = counterClass
                              .getDeclaredConstructor(int.class)
                              .newInstance(NUM_THREADS);

    // Generating an integer array with numbers in the range from LOW to HIGH
    int[] numbers = new int[HIGH-LOW];
    for(int i = 0; i < HIGH - LOW; i++) {
      numbers[i] = LOW + i;
    }

    int result = new SequentialPalindromicCounter().count(numbers);
    long startTime = System.nanoTime();
    int count = counter.count(numbers);
    long endTime = System.nanoTime();
    if (count != result) {
      // Informing when the implementation fails
      System.err.printf("- ERROR: Expected: %d; Found: %d\n", result, count);
    } else {
      // Showing the time taken by the implementation
      System.out.printf("- SUCCESS: Expected: %d; Found: %d"
          + "; Elapsed time: %d ms\n", result, count,
          ((endTime - startTime) / 1000000));
    }
  }

}
```

Please note that you **do not need** to copy source code from this document. The corresponding maven project is attached to this assignment in the folder: **/Ex1**.

While solving this exercise, you are **not allowed** to change the architecture of the system, add or remove any logic that is not explicitly requested. You are only allowed to change what is strictly required in this document.

1.1 Run the project

Run the `Main` application performing the following steps:

1. Open your terminal
2. Navigate to the root folder of the given project, i.e., **/Ex1**
3. Compile the source files:

   ```
   mvn compile
   ```

4. Execute the `Main.main(..)` method:

   ```
   mvn exec:java -Dexec.mainClass="assignment2.Main"
   ```

The following is a sample output produced when `Main.main(..)` is executed. It reports wrong results caused by races.

```
- ERROR: Expected: 19997; Found: 19994
```

Note: since races are non-deterministic, you may need to execute the `Main.main(..)` method multiple times to obtain wrong results as the the ones shown above.

You can test different behaviors changing the constants defined on top of the `Main` class (`LOW`, `HIGH`, and `NUM_THREADS`).

Note: depending on the current VM configuration, a `java.lang.OutOfMemoryError` exception can be thrown when there is insufficient space to allocate new objects in the Java heap. If needed, you can increase the heap space setting the `MAVEN_OPTS` environment variable as follows:

```
export MAVEN_OPTS="-Xms512M -Xmx10G"
```

Where `512M` is an example of a minimum heap size and `10G` is an example of a maximum value set for the heap space.

1.2 Questions

1. Note that class `ThreadPalindromicCounter` produces wrong results. Despite the compact syntax, the operation `total++` (in line 37 of method `ThreadPalindromicCounter.incrementCounter()`) is not an *atomic operation* (i.e., it does not execute as a single, indivisible operation). Instead, this is an example of a *read-modify-write* operation in which the resulting state is derived from the previous state. Since operations in multiple threads may be arbitrarily interleaved, it is possible to encounter races. Without proper synchronization, the results reported by class `ThreadPalindromicCounter` unpredictably depend on scheduling or interleaving, possibly leading to non-atomic updates.

   To make more evident the read-modify-write operation executed in line 37 of method `ThreadPalindromicCounter.incrementCounter()`, it can be replaced with the code shown below:

```
1      int previousTotal = total;
2      total = previousTotal + 1;
```

   Considering that two threads, i.e., `Thread 1` and `Thread 2`, concurrently execute the code above, it is possible to identify each thread/instruction pair by using a letter as following:

	Thread 1	**Thread 2**
1	A	C
2	B	D

   Letter `A` identifies the execution of the first line by `Thread 1`. Next, letter `B` identifies the execution of the second line by `Thread 1`. Similarly, letter `C` and `D` identify the execution of the first and second line by `Thread 2` respectively.

   The table above can be used to exemplify executions as a sequence of letters (as it is shown in slide 19 of the slide set: L3 - Thread Safety). For example:

   - Supposing `total` equals to `9`, an execution `ABCD` leads to `total = 11` which is the expected result.

   Assuming `total = 9`, report a sample execution that leads to a wrong result (i.e., `total != 11` after the sequence is executed). State the sequence of instructions to be executed (using the notation above) and the value of `total` after such sequence is executed.

2. Locate class `SynchronizedMethodsCounter`, a candidate implementation that tries to solve the races previously reported using the `synchronized` keyword to modify the method `PalindromicCounter.run()` as follows:

```
1     public synchronized void run() {
2       for (int i = low; i < high; i++) {
3         if (Palindromic.isPalindromic(nums[i])) {
4           SynchronizedMethodsCounter.this.incrementCounter();
5         }
6       }
7     }
```

Run the `main` method using the `SynchronizedMethodsCounter` class, providing its class name in the `-Dexec.args` option as follows:

```
mvn exec:java -Dexec.mainClass="assignment2.Main"
-Dexec.args="SynchronizedMethodsCounter"
```

You **must not** change the `Main` class, but switch between the different implementations (i.e., `ThreadPalindromicCounter`, `SynchronizedMethodsCounter`, `SynchronizedBlocksCounter`, `VolatileCounter` or `AtomicIntegerCounter`) using the instructions above.

In addition, we provide you JUnit tests to help you test the implementation. There is a unit test for each implementation class. Run the unit test for class `SynchronizedMethodsCounter` using the following command:

```
mvn test -Dtest=MainTest#SynchronizedMethodsCounter
```

The following is a sample output produced when the `MainTest.SynchronizedMethodsCounter()` is executed. It reports the execution and failure of a single test. The reason why the test failed is also reported and highlighted in the following snippet.

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running assignment2.MainTest
[ERROR] Tests run: 1, Failures: 1, Errors: 0, Skipped: 0,
Time elapsed: 2.718 s <<< FAILURE! - in assignment2.MainTest
[ERROR] SynchronizedMethodsCounter  Time elapsed: 2.71 s  <<< FAILURE!
java.lang.AssertionError:

19997 palindromic numbers were expected to be found
between 1 and 99999999, however, 19996 were found.
expected:<19997> but was:<19996>
at assignment2.MainTest.test(MainTest.java:29)
at assignment2.MainTest.SynchronizedMethodsCounter(MainTest.java:39)

[INFO]
[INFO] Results:
[INFO]
[ERROR] Failures:
[ERROR]   MainTest.SynchronizedMethodsCounter:39->test:29
19997 palindromic numbers were expected to be found
between 1 and 99999999, however, 19996 were found.
expected:<19997> but was:<19996>
[INFO]
[ERROR] Tests run:  1, Failures:  1, Errors:  0, Skipped:  0
[INFO]  -------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] -------------------------------------------------------
```

You must always recompile modified source files using the `mvn compile` command (see section 1.1). Only changing the command line parameter, either to run or test an modified implementation, is not enough.

Since races are non-deterministic, you should run the tests multiple times. The presence of a single failing test execution, like the one shown above, confirms the presence of races due to an incorrect implementation.

Explain why the synchronized method does not solve the problem. What does the synchronized keyword inserted in method `run` lock?

3. Modify class `SynchronizedMethodsCounter` using **`synchronized` methods** to avoid the races produced in method `SynchronizedMethodsCounter.incrementCounter()` when multiple threads update the field `total` concurrently.
Note: after that you have modified the implementation, you can compile and run the unit test again using:

```
mvn compile
mvn test -Dtest=MainTest#SynchronizedMethodsCounter
```

A correct implementation should produce an output similar to the following, with zero `Failures`, zero `Errors`, and one `Tests run`:

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running assignment2.MainTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0,
Time elapsed: 6.753 s - in assignment2.MainTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run:  1, Failures:  0, Errors:  0, Skipped:  0
[INFO]
[INFO] -------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] -------------------------------------------------------
```

Please remember that races are non-deterministic and you should run the test multiple times, a single failure indicates a wrong implementation. Please note also that tests are useful only to check whether errors have not been reported. They do not ensure that the exercise has been solved correctly.

4. Locate class `SynchronizedBlocksCounter` (a renamed copy of class `ThreadPalindromicCounter`). Modify this class using **`synchronized` blocks** to avoid the races produced in method `SynchronizedMethodsCounter.incrementCounter()` when multiple threads update the field `total` concurrently.
Update, compile, run, and test the implementation providing `SynchronizedBlocksCounter` as parameter.

5. Locate class `VolatileCounter`, a candidate implementation that tries to solve the race using the `volatile` keyword to modify the `total` field.
   Explain why the `volatile` keyword does not avoid races in class `VolatileCounter`.

```
1 public final class VolatileCounter implements Counter {
2
3   private final int numThreads;
4   private volatile int total;
5
6   private void incrementCounter() {
7     total++;
8   }
```

6. Locate class `AtomicIntegerCounter` (a renamed copy of class `ThreadPalindromicCounter`). Modify the field `total` (currently of type `int`) to be an `AtomicInteger` and implement the suggested changes (pointed by the `TODOs`) to avoid the races produced in method `incrementCounter()` when multiple threads update `total` concurrently.
   Update, compile, run, and test the implementation providing `AtomicIntegerCounter` as parameter.
   Hint: to implement the TODOs, you may need to read the documentation of class AtomicInteger. In particular, you may check the documentation of the constructor and methods incrementAndGet(), get() and set(int).

```java
// TODO 1 : import the java.util.concurrent.atomic.AtomicInteger class

public final class AtomicIntegerCounter implements Counter {
  // TODO 2 : change the type of the "total" field to AtomicInteger
  //          and initialize it to a new AtomicInteger
  private int total;
  private final int numThreads;

  public AtomicIntegerCounter(int numThreads) {
    this.numThreads = numThreads;
  }

  public int count(int[] nums) throws Exception {
    // TODO 3 : call AtomicInteger.set(int value)
    //          to set the current value to 0
    total = 0;

    List<Thread> threads = new LinkedList<Thread>();
    int size = (int) Math.ceil((double) nums.length / numThreads);
    int threadId = 0;
    for (int low = 0; low < nums.length; low += size) {
      PalindromicCounter counter =
            new PalindromicCounter(nums, low, low + size);
      Thread thread = new Thread(counter);
      thread.start();
      threads.add(thread);
    }

    for (Thread thread : threads) {
      thread.join();
    }

    // TODO 4 : call AtomicInteger.get() to get the current int value
    return total;
  }

  private void incrementCounter() {
    // TODO 5 : call AtomicInteger.incrementAndGet()
    //          to atomically increment the current value
    total ++;
  }

  private class PalindromicCounter implements Runnable {
    private final int[] nums;
    private final int low, high;

    private PalindromicCounter(int[] nums, int low, int high) {
      this.nums = nums;
      this.low = low;
      this.high = Math.min(high, nums.length);
    }

    @Override
    public void run() {
      for (int i = low; i < high; i++) {
        if (Palindromic.isPalindromic(nums[i])) {
          AtomicIntegerCounter.this.incrementCounter();
        }
      }
    }
```

```
61          }
62      }
```

7. Locate class `PartialCounter`, a slightly modified copy of the class `ThreadPalindromicCounter`. To reduce contention and increase performance, in this implementation, `PartialCounter.total` is not updated upon the occurrence of each palindromic number. Instead, `PartialCounter.PalindromicCounter.run()` increments a local variable `partialCount` each time a palindromic number is found in the loop. Next, there is a single call to `PartialCounter.addPartialCount(..)` to add the partial result to the `total` (see the code below).
   Aggregating partial results using a local variable makes updates to the shared mutable field less frequent, decreasing the synchronization overhead. However, this modification does not avoid races that can lead to wrong results.
   Modify the field `total` (currently of type `int`) to be an `AtomicInteger` and implement the suggested changes (pointed by the `TODOs`) to avoid the races produced in method `addPartialCount(..)` when multiple threads update `total` concurrently.
   Update, compile, run, and test the implementation providing `PartialCounter` as parameter.
   Hint: to implement the TODOs, you may need to read the documentation of class AtomicInteger. In particular, you may check the documentation of the constructor and methods addAndGet(int), get() and set(int).

```
// TODO 1 : import the java.util.concurrent.atomic.AtomicInteger class

public final class PartialCounter implements Counter {
  // TODO 2 : change the type of the "total" field to AtomicInteger
  //          and initialize it to a new AtomicInteger
  private int total;
  private final int numThreads;

  public PartialCounter(int numThreads) {
    this.numThreads = numThreads;
  }

  public int count(int[] nums) throws Exception {
    // TODO 3 : call AtomicInteger.set(int value)
    //          to set the current value to 0
    total = 0;

    List<Thread> threads = new LinkedList<Thread>();
    int size = (int) Math.ceil((double) nums.length / numThreads);
    int threadId = 0;
    for (int low = 0; low < nums.length; low += size) {
      PalindromicCounter counter =
            new PalindromicCounter(nums, low, low + size);
      Thread thread = new Thread(counter);
      thread.start();
      threads.add(thread);
    }

    for (Thread thread : threads) {
      thread.join();
    }

    // TODO 4 : call AtomicInteger.get() to get the current int value
    return total;
  }

  private void addPartialCount(int partialCount) {
    // TODO 5 : call AtomicInteger.addAndGet(int delta)
    //          to atomically add partialCount to the current value
    total += partialCount;
  }
```

```
42
43    private class PalindromicCounter implements Runnable {
44      private final int[] nums;
45      private final int low, high;
46
47      private PalindromicCounter(int[] nums, int low, int high) {
48        this.nums = nums;
49        this.low = low;
50        this.high = Math.min(high, nums.length);
51      }
52
53      @Override
54      public void run() {
55        int partialCount = 0;
56        for (int i = low; i < high; i++) {
57          if (Palindromic.isPalindromic(nums[i])) {
58            partialCount++;
59          }
60        }
61        PartialCounter.this.addPartialCount(partialCount);
62      }
63    }
64  }
```

Exercise 2 - Ticket Shop - Thread Safety (3 points)

A ticket shop sells tickets for a certain event. Each ticket has a unique ID number and guarantees admission to the event to a single customer. For convenience, tickets are not sold directly by the shop but from certified sellers which take them from a unique and centralized system managed by the shop. A seller can manage only a customer at a time. Multiple sellers can sell tickets in parallel.

The following is the Java implementation of class `Ticket`. Each ticket has a unique ID, generated using a counter.

```
1  public final class Ticket {
2
3    private static long count = 0;
4    private final long id;
5
6    public Ticket() {
7      this.id = count++;
8    }
9
10   public long getId() {
11     return id;
12   }
13
14 }
```

A ticket shop must implement the `TicketShop` interface. In this way, it is possible to easily test multiple and different implementations.

```
1  public interface TicketShop {
2
3    Ticket buy() throws TicketsSoldOutException;
4
5  }
```

`TicketShoplImpl` is an initial implementation of interface `TicketShop`. It handles the available tickets and retrieves them on request.
When created, the ticket shop instantiates a certain number of tickets, which is provided as a parameter (`numberOfTickets`) and puts such tickets into an array. When calling method `buy` it returns the first available ticket, incrementing the `nextTicket` index. If no ticket is available, a request to buy throws a `TicketsSoldOutException`.

```
public final class TicketShopImpl implements TicketShop {

  private final Ticket[] availableTickets;
  private int nextTicket;

  TicketShopImpl(int numberOfTickets) {
    this.availableTickets = new Ticket[numberOfTickets];

    for (int i = 0; i < numberOfTickets; i++) {
      availableTickets[i] = new Ticket();
    }
  }

  public Ticket buy() throws TicketsSoldOutException {
    Ticket ticket = null;

    if (nextTicket < availableTickets.length) {
      ticket = availableTickets[nextTicket];
      nextTicket += 1;
    } else {
      throw new TicketsSoldOutException();
    }

    processOrder(ticket);

    return ticket;
  }

  public void processOrder(Ticket ticket) {
    // Simulate ticket elaboration with waiting time
    try {
      Thread.sleep(1);
    } catch (Exception ex) {}
  }

}
```

`Customers` are uniquely identified and able to buy and hold a single ticket. Sellers are able to call their `buyTicket` method indicating the reference shop.

```
public final class Customer {
  private final long id;
  private Ticket ticket;

  public Customer(long id) {
    this.id = id;
  }

  public void buyTicket(TicketShop ticketShop)
  throws TicketsSoldOutException {
    this.ticket = ticketShop.buy();
  }

  public long getId() {
    return id;
  }

  public Ticket getTicket() {
    return ticket;
  }
}
```

Sellers are able to handle a certain number of customers and buy a ticket for them, one at a time. In practice, when creating a `Seller`, `customers` have to be provided as an array, with two indices, `low` and `high`, to address only a sub-portion of such array. The reference `TicketShop` needs also to be provided. The `run` method is able to call the previously defined `buyTicket` providing a `Ticket` for each `Customer`.

```
public final class Seller implements Runnable  {

  private final Customer[] customers;
  private final int low, high;
  private final TicketShop ticketShop;

  public Seller(
    Customer[] customers,
    int low,
    int high,
    TicketShop ticketShop
  ) {
    this.customers = customers;
    this.low = low;
    this.high = Math.min(high, customers.length);
    this.ticketShop = ticketShop;
  }

  @Override
  public void run() {
    for (int i = low; i < high; i++) {
      try {
        customers[i].buyTicket(ticketShop);
      } catch(TicketsSoldOutException ex) {
        System.out.println(
          "Cannot buy ticket for customer " + customers[i].getId() +
          ". Shop out of stock."
        );
      }
    }
  }
}
```

A test scenario has been implemented in the `Main` class. It allocates a `TicketShop`, several `Customers`, and `Sellers` making them communicate with each other. Please note that `Sellers` are actually `Runnable`, so the number of sellers represents the number of `Threads`. All these threads share the same ticket shop instance, using the same ticket stock. At the end of the execution, the program performs some sanity checks to ensure the correctness of the implementation, printing the errors (if any) on the standard error output.

```java
public final class Main {

  private static final int NUMBER_OF_SELLERS = 8;
  private static final int NUMBER_OF_TICKETS = 10_000;
  private static final int NUMBER_OF_CUSTOMERS = 10_000;

  private static final int CUSTOMERS_PER_SELLER = (int) Math.ceil(
    (double) NUMBER_OF_CUSTOMERS / NUMBER_OF_SELLERS
  );

  public static void main(String[] args) throws Exception  {
    Class<? extends TicketShop> ticketShopClass = null;

    String ticketShopClassName = args.length > 0
      ? args[0] :
      "TicketShopImpl";
    try {
      ticketShopClass = Class.forName("assignment2." + ticketShopClassName)
                             .asSubclass(TicketShop.class);
    } catch (ClassNotFoundException ex) {
      System.err.println(
        "- ERROR: the provided command line parameter is not valid. " +
        "Cannot find a class named \"" + ticketShopClassName + "\""
      );
      return;
    }

    Thread[] threads = new Thread[NUMBER_OF_SELLERS];
    TicketShop ticketShop = ticketShopClass
                                  .getDeclaredConstructor(int.class)
                                  .newInstance(NUMBER_OF_TICKETS);
    Customer[] customers = new Customer[NUMBER_OF_CUSTOMERS];

    for (int i = 0; i < NUMBER_OF_CUSTOMERS; i++) {
      customers[i] = new Customer(i);
    }

    // Sellers work in parallel
    for (int i = 0; i < NUMBER_OF_SELLERS; i++) {
      int start = i * CUSTOMERS_PER_SELLER;
      int end = Math.min(start + CUSTOMERS_PER_SELLER, customers.length);

      Seller seller = new Seller(customers, start, end, ticketShop);
      threads[i] = new Thread(seller);
    }

    long startTime = System.nanoTime();

    for (Thread thread : threads) {
      thread.start();
    }

    for (Thread thread : threads) {
      thread.join();
    }

    long stopTime = System.nanoTime();

    // Assert that the same ticket has been sold only once
    Map<Ticket, Integer> duplicates = new HashMap<Ticket, Integer>();
```

```
61      for (int i = 0; i < customers.length; i++) {
62        Customer customer = customers[i];
63        Ticket ticket = customer.getTicket();
64
65        if (ticket != null) {
66          int numberOfOccurences = duplicates.getOrDefault(ticket, 0);
67          duplicates.put(ticket, numberOfOccurences + 1);
68        }
69      }
70
71      for(Map.Entry<Ticket, Integer> entry : duplicates.entrySet()) {
72        Ticket ticket = entry.getKey();
73        int times = entry.getValue();
74
75        if (times > 1) {
76          System.err.println(
77            "- ERROR: Unique Ticket number " + ticket.getId() +
78            " has been sold " + times + " times."
79          );
80        }
81      }
82
83      System.out.println(
84        "\nExecution time: " + (
85          TimeUnit.MILLISECONDS.convert(stopTime-startTime, TimeUnit.NANOSECONDS)
86        ) + " ms"
87      );
88    }
89
90  }
```

Please note that you **do not need** to copy the source code from this document. The maven project is attached to this assignment in the folder: **/Ex2**.

You are **not allowed** to change the architecture of the system, add or remove any logic that is not explicitly requested. You are allowed to change what strictly needed to solve the exercise.

You can test different behaviors changing the constants defined on top of the `Main` class (`NUMBER_OF_SELLERS`, `NUMBER_OF_TICKETS`, and `NUMBER_OF_CUSTOMERS`).

2.1 Questions

1. Consider the implementation provided in the `TicketShopImpl` class. Using the commands defined in 1.1, the following is a sample output obtained after running `Main.main(..)`:

```
- ERROR: Unique Ticket number 8944 has been sold 4 times.
- ERROR: Unique Ticket number 3442 has been sold 4 times.
- ERROR: Unique Ticket number 6058 has been sold 2 times.
- ERROR: Unique Ticket number 7113 has been sold 2 times.
- ERROR: Unique Ticket number 9369 has been sold 3 times.
- ERROR: Unique Ticket number 6623 has been sold 2 times.
- ERROR: Unique Ticket number 7811 has been sold 2 times.
- ERROR: Unique Ticket number 8367 has been sold 2 times.
- ERROR: Unique Ticket number 8398 has been sold 4 times.
- ERROR: Unique Ticket number 6707 has been sold 2 times.
- ERROR: Unique Ticket number 7972 has been sold 2 times.
- ERROR: Unique Ticket number 3535 has been sold 4 times.

Execution time: 1655 ms
```

   This means that a single ticket, identified by its ID, has been sold multiple times to different customers, which should not happen. The error can be identified in the `Ticket buy()` method of the `TicketShopImpl` class, since the implementation lacks proper synchronization, races might happen based on the scheduling or interleaving of multiple concurrent threads. In particular, it is needed to synchronize the accesses to the `availableTickets` array using the index `nextTicket`, from line `19` to line `21`.

```
1       if (nextTicket < availableTickets.length) {
2         ticket = availableTickets[nextTicket];
3         nextTicket += 1;
```

   Considering two concurrent threads, `Thread 1` and `Thread 2`, it is possible to identify each thread/instruction pair by using a letter as following:

	Thread 1	**Thread 2**
1	A	D
2	B	E
3	C	F

   Using the notation already introduced in the first exercise, report two example executions that lead to two different kinds of error, i.e., the race reported above and an `Exception`.

   Note: if you are using an IDE when running `Main.main(..)` or executing tests, please note that the Console view currently cannot ensure that colored mixed standard and error output is shown in the same order as it is produced by the running process. As a result, the output obtained when executing `Main.main(..)` may appear in a different order and this does not indicate any erroneous behavior in the application.

   Hint: the program can crash under a specific condition and with a specific execution order. Consider different states (i.e., different values stored in `availableTickets` and `nextTicket` fields) and execution orders. You **do not need** to change the parameters defined on top of the `Main` class (`NUMBER_OF_SELLERS`, `NUMBER_OF_TICKETS`, and `NUMBER_OF_CUSTOMERS`).

2. Locate class `TicketShopSynchronizedMethods` (a renamed copy of class `TicketShopImpl`). Modify this class using **`synchronized` methods** to avoid the races produced in method `TicketShopSynchronizedMethods.buy()` when multiple threads access and update fields `availableTickets` and `nextTicket` concurrently.
   Update, compile, run, and test the implementation providing `TicketShopSynchronizedMethods` as parameter.

3. Locate class `TicketShopSynchronizedBlocks` (a renamed copy of class `TicketShopImpl`). Modify this class using **`synchronized` blocks** to avoid the races produced in method `TicketShopSynchronizedBlocks.buy()` when multiple threads access and update fields `availableTickets` and `nextTicket` concurrently.
   Please make sure that the synchronized block encloses only what is strictly needed, i.e., do not use synchronization for code portions that do not need it.
   Update, compile, run, and test the implementation providing `TicketShopSynchronizedBlocks` as parameter.

4. Compare the performance of the two previous solutions (`TicketShopSynchronizedMethods` and `TicketShopSynchronizedBlocks`) executing the `Main.main(..)` method, report the results in terms of execution time, and explain why one performs better than the other.
   Hint: consider the critical sections of the two implementations.
   Note: the execution time is automatically printed at the end of the execution. You do not need to develop additional code. Note that the reported elapsed time depends on the CPU, number of threads/cores available, other processes running concurrently, etc.

5. Locate class `TicketShopStack` (a renamed copy of class `TicketShopImpl`). Modify this class replacing the `availableTickets` and `nextTicket` fields (currently of type `Ticket[]` and `int`) to be a `Stack<Ticket>` and implement the suggested changes (pointed by the `TODOs`) to avoid the races produced in method `buy()` when multiple threads access and update them concurrently.
   Update, compile, run, and test the implementation providing `TicketShopStack` as parameter.
   Hint: to implement the TODOs, you may need to read the documentation of class Stack. In particular, you may check the documentation of the constructor and methods push(E) and pop().

```
1  // TODO 1 : import the java.util.EmptyStackException class
2  // TODO 2 : import the java.util.Stack class
3
4  public final class TicketShopStack implements TicketShop {
5    // TODO 3 : Replace the following fields with a single Stack<Ticket>
6    private final Ticket[] availableTickets;
7    private int nextTicket;
8
9    TicketShopStack(int numberOfTickets) {
10     // TODO 4 : Replace the following initializations
11     //          with the one of the Stack
12     this.availableTickets = new Ticket[numberOfTickets];
13
14     // TODO 5 : push tickets onto the top of the Stack
15     //          calling Stack.push(Ticket ticket)
16     for (int i = 0; i < numberOfTickets; i++) {
17       availableTickets[i] = new Ticket();
18     }
19   }
20
21   public Ticket buy() throws TicketsSoldOutException {
22     Ticket ticket = null;
23
24     // TODO 6 : Replace the following logic calling Stack.pop()
25     //          to remove and return the ticket at the top of the stack,
26     //          assigning it to the "ticket" variable.
27     //          You would need to catch the EmptyStackException.
28     //          Please do not change the behavior of the program,
29     //          if no ticket is available raise a
30     //          "TicketsSoldOutException" exception.
31     if (nextTicket < availableTickets.length) {
32       ticket = availableTickets[nextTicket];
33       nextTicket += 1;
34     } else {
35       throw new TicketsSoldOutException();
36     }
37
38     processOrder(ticket);
39
40     return ticket;
41   }
42
43   public void processOrder(Ticket ticket) {
44     // Simulate ticket elaboration with waiting time
45     try {
46       Thread.sleep(1);
47     } catch (Exception ex) {}
48   }
49
50 }
```

6. Referring to the `TicketShopStack` class, does the usage of the Java `Stack` class solve the races? Justify your answer.
   Hint: `Stack<E>` extends `Vector<E>`, read carefully its the Java documentation, in particular, the explanation regarding thread-safety (see the summary on top of the page).

SUBMISSION INSTRUCTIONS

(please notice that the following instructions are mandatory. Submissions that do not comply, will not be graded):

- Assignments are individual work. Duplicates are not graded.
- Assignments must be submitted via **iCorsi**. Do not send anything by email.
- Solutions to theoretical questions must be presented inside a (single) PDF or plain text file named:
  assignment<AssignmentNumber>.<Surname>.<Name><.pdf|.txt>
  In that document you must clearly specify your name and the question you are addressing. Please do not submit scanned documents using handwritten text.
- In addition, you must attach the Java sources that you have completed to solve the assignment. Submissions without Java source code or with Java source code that does not compile will not be graded.
- All the produced files must be collected into a single archive file (.zip or .tar) named:
  assignment<AssignmentNumber>.<Surname>.<Name><.zip|.tar.gz>