Programming Fundamentals 3 **2020**

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

---

Assignment 4 (12 points) **Due date: 13 December 2020, 08:00 a.m.**

For questions regarding this assignment, please contact Filippo Schiavio via email: filippo.schiavio@usi.ch.

This assignment contributes 12% to the overall grade. Please follow strictly the submission instructions written at the end of the assignment.

Environment Setup

The exercises presented in this assignment are available as a maven project in the attached folder: **/ExGaming**. Before proceeding, you are advised to properly set the JAVA_HOME variable and you must install maven following the official installation guide.

Run the Project

To compile and run the project, you should use the same instructions provided in the assignments 2 and 3. For instance:

```
# compile the java source
mvn compile
# run the main class of an exercise, e.g., exercise 1a:
mvn exec:java@ex1a
```

If you need further details or you encounter errors, please refer to the Java Tools lecture recordings, which include a detailed explanation of maven and a step by step guide to install and run it.

Please note that we do not provide JUnit tests for this assignment; however, you can check your solution by executing the simulation provided for each exercise. In case of an error, the simulation terminates, also printing the error information to the console.

Note: please remember that races are non-deterministic such that you should run the provided simulations multiple times. A single failure indicates a wrong implementation, however, the absence of failures does not necessarily ensure that an exercise was solved correctly.

Exercise 1 - Gaming Catalog (9 points)

An online gaming platform offers a catalog of trial games. Developers add games to the catalog while gamers play the available games. The catalog has a fixed size and is full of games by default. When a developer wants to add a new game, the oldest one is replaced, i.e., removed from the catalog. A game that has been replaced becomes unavailable and gamers must not be able to play that game anymore. A lot of gamers exploit the gaming platform while developers rarely update the catalog, meaning that read accesses are far more frequent than writing ones.

The following is the Java implementation of class `Game`. Each game has the boolean field `available`, that indicates whether it is available in the catalog and thus can be played. If a gamer tries to play a game that is not available, a `GameNotAvailableException` is thrown.

```
1  public final class Game {
2    private boolean available = true;
3
4    public void setAvailable() {
5      this.available = true;
6    }
7
8    public void setUnavailable() {
9      this.available = false;
10   }
11
12   public void play() throws InterruptedException, GameNotAvailableException {
13     if (!available) {
14       throw new GameNotAvailableException();
15     }
16     Thread.sleep(10); // Simulate playing time
17   }
18 }
```

A catalog must implement the `Catalog` interface. In this way, it is possible to easily test multiple and different implementations of a catalog.

```
1  public interface Catalog {
2    void publish(Game newGame) throws InterruptedException;
3    void play(int index) throws InterruptedException, GameNotAvailableException;
4    int size();
5  }
```

`CatalogIImpl` is an initial implementation of interface `Catalog`. Upon creation, a number of games are added to fill the new catalog. Method `publish(..)` enables developers to publish new games while method `play(..)` enables gamers to play games. Games are stored into the `games` array field and replaced starting from the beginning of the array, using the `gameToReplace` index. The catalog is also in charge of switching the availability of a game, calling `Game.setUnavailable()` when a game is replaced.

```
1  public class CatalogImpl implements Catalog {
2    private final Game[] games;
3    private int gameToReplace = 0;
4
5    public CatalogImpl(int numberOfGames) {
6      games = new Game[numberOfGames];
7      for (int i = 0; i < games.length; i++) {
8        games[i] = new Game();
9      }
10   }
11
12   public int size() {
13     return games.length;
14   }
15
16   public void publish(Game newGame) throws InterruptedException {
17     games[gameToReplace].setUnavailable();
18     Thread.sleep(10);
19     games[gameToReplace] = newGame;
20     Thread.sleep(10);
21     gameToReplace++;
22     Thread.sleep(10);
23     if (gameToReplace >= games.length) {
24       gameToReplace = 0;
25     }
26     newGame.setAvailable();
27   }
28
29   public void play(int index) throws InterruptedException,
30       GameNotAvailableException {
31     games[index].play();
32   }
33 }
```

Note that the class `CatalogImpl` is not thread safe, in particular, the following issues could take place:
- If two threads execute the method `CatalogImpl.publish` there could be a race on the field `CatalogImpl.gameToReplace`, leading to an ArrayIndexOutOfBoundsException.

- If one thread executes the method `CatalogImpl.publish` and another thread executes the method `CatalogImpl.play`, the second thread could attempt to play a game which has been set to unavailable, leading to a `GameNotAvailableException`.

Such issues must be solved introducing proper synchronization in the catalog, e.g., using synchronized methods as shown below in the provided class `ex1.CatalogSynchronized`.

```
1  public class CatalogSynchronized extends CatalogImpl {
2
3    public CatalogSynchronized(int numberOfGames) {
4      super(numberOfGames);
5    }
6
7    public synchronized void publish(Game newGame) throws InterruptedException {
8      super.publish(newGame);
9    }
10
11   public synchronized void play(int index) throws InterruptedException,
12       GameNotAvailableException {
13     super.play(index);
14   }
15 }
```

Developers are able to publish a single game by calling the method `Catalog.publish(..)`.

```
public final class Developer implements Runnable {
  private final Catalog catalog;

  public Developer(Catalog catalog) {
    this.catalog = catalog;
  }

  @Override
  public void run() {
    try {
      Thread.sleep(10);
      catalog.publish(new Game());
    } catch (ArrayIndexOutOfBoundsException ex) {
      System.err.println("- ERROR: Race between developers");
      System.exit(1);
    } catch (InterruptedException ex){
      System.err.println("InterruptedException not supported");
      System.exit(2);
    }
  }
}
```

Gamers are able to play a single and random game by calling the `Catalog.play(..)`.

```
public class Gamer implements Runnable {
  private final Catalog catalog;

  public Gamer(Catalog catalog) {
    this.catalog = catalog;
  }

  @Override
  public void run() {
    try {
      catalog.play((int) (Math.random() * catalog.size()));
    } catch (GameNotAvailableException ex) {
      System.err.println("- ERROR: Gamer trying to play an unavailable game");
      System.exit(1);
    } catch (InterruptedException ex){
      System.err.println("InterruptedException not supported");
      System.exit(2);
    }
  }
}
```

The provided classes `Developer` and `Gamer` check that the races mentioned above do not take place, if so, an error message is printed to the standard error and the application terminates.

The `Simulator` abstract class implements the core logic for simulating the behaviour of multiple developers and gamers. The simulation allocates a `Catalog` and schedules different `Developers` and `Gamers` runnable tasks, then it starts them, waits for their completion and prints the total execution time.

```
public abstract class Simulator {
  protected final int nDevelopers = Parameters.getNumberDevelopers();
  protected final int nGamers = Parameters.getNumberGamers();

  public final void run() throws InterruptedException {
    System.out.println("Simulation: #dev="+nDevelopers+" #gamers="+nGamers);
    Catalog catalog = CatalogFactory.create();
    for (int i = 0; i < nGamers; i++) {
      addGamer(new Gamer(catalog));
    }
    for (int i = 0; i < nDevelopers; i++) {
      addDeveloper(new Developer(catalog));
    }
    long start = System.nanoTime();
    startWorkers();
    waitWorkers();
    long end = System.nanoTime();
    long execTime = MILLISECONDS.convert(end - start, NANOSECONDS);
    System.out.println("Execution time: " + execTime + " ms");
  }

  protected abstract void addDeveloper(Developer developer);
  protected abstract void addGamer(Gamer gamer);

  protected abstract void startWorkers();
  protected abstract void waitWorkers() throws InterruptedException;
}
```

A simulation scenario has been implemented in the `Ex1aSimulator` class, which keeps a list of threads, populated in the methods `addDeveloper` and `addGamer`, starts them in the method `startWorkers` and waits their completion in the method `waitWorkers` by leveraging the method `Thread.join`.

```
class Ex1aSimulator extends Simulator {

  public static void main(String[] args) throws InterruptedException {
    new Ex1aSimulator().run();
  }

  private final List<Thread> threads = new LinkedList<>();

  @Override
  public void addDeveloper(Developer developer) {
    threads.add(new Thread(developer));
  }

  @Override
  public void addGamer(Gamer gamer) {
    threads.add(new Thread(gamer));
  }

  @Override
  protected void startWorkers() {
    Collections.shuffle(threads);
    for(Thread thread : threads) {
      thread.start();
    }
  }

```

```
27    @Override
28    protected void waitWorkers() throws InterruptedException {
29      for(Thread thread : threads) {
30        thread.join();
31      }
32    }
33  }
```

You can execute the simulation scenario by specifying the `Catalog` implementation (e.g., `CatalogImpl`) with the following command:
`mvn exec:java@ex1a -DcatalogClass=ex1.CatalogImpl`

Since the class `CatalogImpl` is not thread safe, while executing the simulation with that catalog implementation you would get an error similar to the one below:

```
Start Simulation: #dev=200 #gamers=1000
- ERROR: Gamer trying to play an unavailable game
```

On the other hand, the simulation runs correctly if executed using the `CatalogSynchronized` implementation, as it is thread safe. Unfortunately, the `CatalogSynchronized` implementation exhibits poor performance. Throughout this exercise you will be asked to improve the catalog and the simulation implementations.

Please note that you **do not need** to copy the source code from this document. The corresponding maven project is attached to this assignment in the folder: **/ExGaming**. You are **not allowed** to change the architecture of the system, add or remove any logic that is not explicitly requested in this document to solve the exercise.

1.a Improving the Catalog - ReentrantReadWriteLock

As mentioned before, `CatalogSynchronized` is thread safe and implements a mutual-exclusion lock, i.e., at most one thread at a time can hold a lock. However, mutual exclusion is frequently a stronger locking discipline than needed to preserve data integrity, and thus may limit concurrency more than necessary. For instance, mutual exclusion prevents *writer/writer* and *writer/reader* overlap, but also prevents *reader/reader* overlap. In many cases, data structures need to be modified, but most accesses involve only reading. In these cases, *read-write locks* allow a shared data structure to be accessed by multiple readers or a single writer, but not both at a time.
Locate the class `ex1/CatalogReadWriteLock` (a subclass of `CatalogImpl`). Modify this class using a java.util.concurrent.locks.ReentrantReadWriteLock to avoid the races mentioned before, but allowing multiple gamers to play concurrently.

```
1  public final class CatalogReadWriteLock extends CatalogImpl {
2
3    public CatalogReadWriteLock(int numberOfGames) {
4      super(numberOfGames);
5    }
6
7    public void publish(Game newGame) throws InterruptedException {
8      super.publish(newGame);
9    }
10
11   public void play(int index) throws InterruptedException,
12       GameNotAvailableException {
13     super.play(index);
14   }
15 }
```

Please note that the application uses, by default, the class `CatalogReadWriteLock` as catalog implementation, so you can execute the simulation using that implementation with the commands:

```
mvn compile
mvn exec:java@ex1a
```

1.b Improving the Threads Harness - CountDownLatch

The `Simulator` class aims to collect the execution time of the worker threads (i.e., developers and gamers). With the implementation `Ex1aSimulator` used so far, the time taken for the creation and termination of the threads is part of the collected execution time. In contrast, it would be desired to collect only the time spent executing the tasks. Such a behavior could be implemented by starting the timer once the threads are ready to execute the `run` method and stopping the timer once they all have executed the `run` method. This implementation requires that the runnables executed by each thread can be monitored, i.e., before executing the `run` method they should wait for a signal from the main thread, notifying that all threads have been started. Moreover, the threads should send a completion signal to the main thread, so that it can stop the timer once all completion signals have been received. In this exercise you have to implement the described behavior using latches (i.e., instances of the class java.util.concurrent.CountDownLatch).
Locate the provided class `ex1/Ex1bSimulator` (listed below), in such class we provide also an helper class `MonitoredRunnable`, which wraps a runnable instance (either developer or gamer). You should use that class to implement the described logic for monitoring the execution of the `run` method. We also provide the implementation of methods `addDeveloper` and `addGamer`, which wrap the runnable using class `MonitoredRunnable` and starts a thread with the wrapper instance as runnable. Note that **you are not allowed to modify** methods `addDeveloper` and `addGamer`.

In this exercise, you have to implement the TODOs provided in the code, implementing the logic for sending and waiting the signals described above. Please note that there is no automatic error checking for this exercise.
You can execute this exercise with the commands:

```
mvn compile
mvn exec:java@ex1b
```

Please note that, as mentioned before, the catalog implementation class used by default is `CatalogReadWriteLock`, which you should have correctly implemented in the previous exercise.

```java
public class Ex1bSimulator extends Simulator {

  public static void main(String[] args) throws InterruptedException {
    new Ex1bSimulator().run();
  }

  // TODO: instantiate the required latches. Note that you have access
  // to the protected fields nDevelopers and nGamers in the super class

  protected class MonitoredRunnable implements Runnable {
    private final Runnable runnable;

    public MonitoredRunnable(Runnable runnable) {
      this.runnable = runnable;
    }

    @Override
    public void run() {
      try {
        // TODO: wait a start signal from the main thread

        runnable.run();
        // TODO: signal the main thread that the execution completed

      } catch (Exception e) {
        System.out.println("Exception not supported");
        System.exit(1);
      }
    }
  }

  @Override
  protected void startWorkers() {
    // TODO: signal the worker threads that they can proceed
  }

  @Override
  protected void waitWorkers() throws InterruptedException {
    // TODO: wait for the workers to complete
  }

  @Override
  protected void addDeveloper(Developer developer) {
    new Thread(new MonitoredRunnable(developer)).start();
  }

  @Override
  protected void addGamer(Gamer gamer) {
    new Thread(new MonitoredRunnable(gamer)).start();
  }
}
```

1.c Improving the Tasks Execution - ThreadPool

The simulator implementations used so far create a thread for each task (either developer or gamer) to be executed. However, this approach has a serious drawback. If the simulation needs to execute many tasks, then the application would gradually become less responsive, up to the point where the system reactivity can be seriously undermined, as the overhead of all those threads exceeds the capacity of the system.
In this exercise you have to solve the mentioned issue by executing tasks in a thread pool, instead of spawning one thread for each task, as done in the previous simulations. Locate the class `ex1/Ex1cSimulator`, a subclass of `ex1/Ex1bSimulator`. You should implement the TODOs in the provided code. In particular, you should create two instances of thread pools, one for developers and one for gamers. Both thread pools should be instantiated with a fixed number of threads, which should be equal to the number of available processors on the system. Note that **you should not** explicitly code the number of processors in your actual machine, instead, you should obtain the number of available processors on the system programmatically, relying on the java.lang.Runtime class. Please note that there is no automatic error checking for this exercise.

You can execute this exercise with the commands:

```
mvn compile
mvn exec:java@ex1c
```

```java
public class Ex1cSimulator extends Ex1bSimulator {

  public static void main(String[] args) throws InterruptedException {
    new Ex1cSimulator().run();
  }

  // TODO: get the number of available processors (N) in your system and
  // instantiate two thread pools, one for developers and one for
  // gamers, both of them with N threads

  @Override
  protected void addDeveloper(Developer developer) {
    Runnable runnable = new MonitoredRunnable(developer);
    // TODO: execute the runnable in the developers pool
  }

  @Override
  protected void addGamer(Gamer gamer) {
    Runnable runnable = new MonitoredRunnable(gamer);
    // TODO: execute the runnable in the gamers pool
  }

  @Override
  protected void waitWorkers() throws InterruptedException {
    // TODO: shutdown the thread pools
    super.waitWorkers();
  }
}
```

Exercise 2 - Gaming Arena - CyclicBarrier (3 points)

In this exercise you have to deal with a different gaming platform, i.e., an arena which allows multiple gamers to play a list of games. Each gamer should play all the games in the list and in the provided order. Moreover, each game should be played exactly once, then the gamer should play the next game, if any. The arena establishes the following protocol. First, the number of gamers that can play in the arena is fixed and known in advance, i.e., gamers are not allowed to enter the arena once gaming has already started, or leave the arena before completing all the games. Moreover, it is required that each gamer waits for all other gamers to be ready for playing the i-th game before playing it, i.e., two gamers must not play different games at the same time.

The provided code for this exercise is located in a single file `ex2/Ex2Simulator.java`, shown below. You should ensure the described protocol using a barrier, i.e., an instance of class java.util.concurrent.CyclicBarrier, for waiting the other gamers. The simulator takes care of initializing an array of games, setting all of them as unavailable. We provide the `Runnable` class `GamersReady` for taking care of making the i-th game available. Please note that **you are not allowed** to directly invoke the method `Game.setAvailable` in your code. Instead, you have to use the provided class `GamersReady`, ensuring that the method `GamersReady.run` is called exactly once for each iteration in the loop executed by the gamers, when all gamers are ready to play the i-th game, i.e., the method `GamersReady.run` has to be called by the last thread that reaches the barrier. You should ensure that, before the gamers can play the current game, the method `GamersReady.run` completes and its effects are visible to all the gamer threads.

You can execute this exercise with the commands:

```
mvn compile
mvn exec:java@ex2
```

```java
public class Ex2Simulator {

  public static void main(String[] args) {
    new Ex2Simulator().run();
  }

  private final int nGamers = Parameters.getNumberGamers();
  private final Game[] games = new Game[Parameters.getNumberGames()];
  // TODO: instantiate a CyclicBarrier instance

  public Ex2Simulator() {
    for (int i = 0; i < games.length; i++) {
      games[i] = new Game();
      games[i].setUnavailable();
    }
  }

  private class ArenaGamer implements Runnable {
    @Override
    public void run() {
      try {
        for (int i = 0; i < games.length; i++) {
          // TODO: wait the other gamers before playing the i-th game.
          //       The last thread that gets ready should take care
          //       of making the i-th game available
          games[i].play();
        }
      } catch (GameNotAvailableException ex) {
        System.err.println("ERROR: Gamer trying to play an unavailable game");
        System.exit(1);
      } catch (Exception e) {
        System.out.println("Exception not supported");
        System.exit(2);
      }
    }
  }

  private class GamersReady implements Runnable {
    private int currentGame = 0;

    @Override
    public void run() {
      try {
        Thread.sleep(10);
      } catch (InterruptedException ex) {
        System.out.println("InterruptedException not supported");
        System.exit(1);
      }
      games[currentGame++].setAvailable();
      System.out.println("Gamers are ready to play the game #" + currentGame);
    }
  }

  public void run() {
    for (int i = 0; i < nGamers; i++) {
      new Thread(new ArenaGamer()).start();
    }
  }
}
```

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>