Practicum OO – lente 2016 **Opgave 14**

Goals

This exercise consists of three programs with threads. Use the *6 steps approach to designing concurrent programs* outlined in the lectures and available on BlackBoard. After this exercise you should be able to:

- synchronize threads;
- stop threads;
- use the `Worker` interface and the class `Task`.

1 Simulating trains and taxis transporting people to and from a station

Introduction

In this exercise we will simulate the process of arrivals of a train with persons and the transportation of them to a certain location by taxis. We start with a sequential solution which must be changed to a concurrent version in the assignment.

We will use simple classes so the focus is placed on the process of transporting itself. A train transports a number of persons (between 60 and 90) to a station. These persons must be transported to a holiday location at some distance. Four taxis are available for the transport: two with a capacity of four persons and two with a capacity of seven persons.

There are 10 arrivals of the train and the simulation stops when all the passengers have been transported.

Design and implementation of the sequential version

The class diagram of figure 1 shows the classes that has been used.

The code of method `main` of class `Main` reads:

```
package taxirides;

public class Main {
  public static void main(String[ ] args) {
    Simulation sim = new Simulation();
    while (!sim.ended()) {
      sim.step();
    }
    sim.showStatistics();
  }

}
```

Class Simulation has this code:

```
package taxirides;

public class Simulation {

  public static final int TRAIN_TRIPS = 10;
  public static final int MIN_TRAVELLERS = 60;
  public static final int MAX_TRAVELLERS = 90;
```

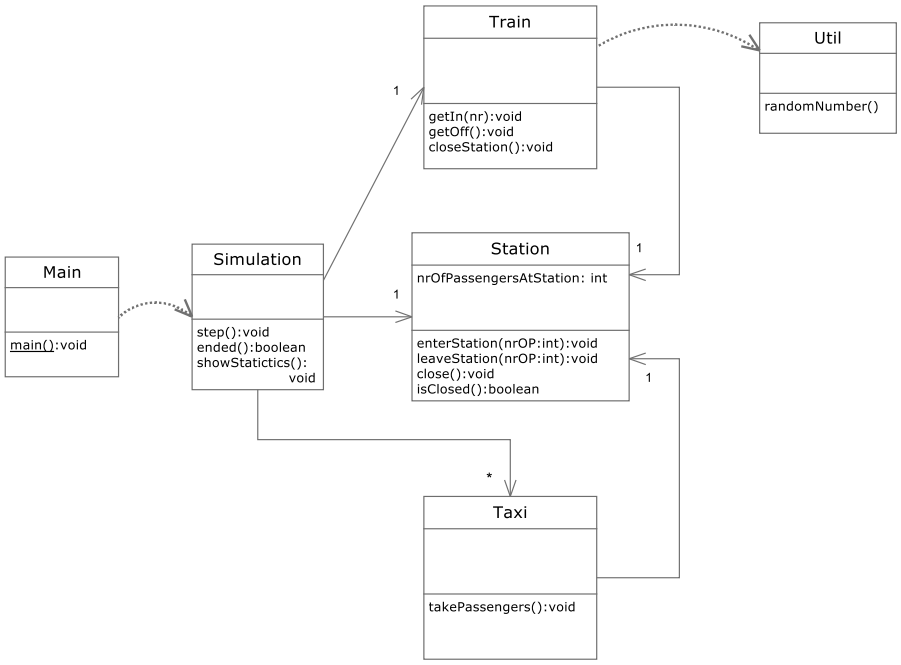

Figure 1: Class Diagram of the sequential version.

```
public static final int CAPACITYSMALL = 4;
public static final int CAPACITYLARGE = 7;
public static final int TIMESMALL = 2;
public static final int TIMELARGE = 3;
public static final int NROFTAXIS = 4;
public static final int NROFSMALLTAXIS = 2;

private Taxi[ ] taxis;
private Train train;
private Station station;

private boolean hasEnded = false;
private int nextTaxi = 0;

public Simulation() {
  station = new Station();
  taxis = new Taxi[NROFTAXIS];
  for (int i = 0; i < NROFTAXIS; i++) {
    taxis[i] =
        i < NROFSMALLTAXIS ?
            new Taxi(i + 1, CAPACITYSMALL, TIMESMALL, station) :
            new Taxi(i + 1, CAPACITYLARGE, TIMELARGE, station);
  }
  train = new Train(station);
}

public void step() {
  if (station.getNrOfPassengersWaiting() > 0) {
    taxis[nextTaxi].takePassengers();
    nextTaxi = (nextTaxi + 1) % NROFTAXIS;
  }
```

```
    else if (train.getNrOfTrips() < TRAIN_TRIPS) {
      train.getIn(Util.getRandomNumber(MIN_TRAVELLERS, MAX_TRAVELLERS));
      train.getOff();
    }
    else {
      train.closeStation();
      hasEnded = true;
    }
  }

  public boolean ended() {
    return hasEnded;
  }

  public void showStatistics() {
    System.out.println("All persons have been transported");
    System.out.println("Total time of this simulation:" + calcTotalTime(taxis));
    System.out.println("Total nr of train travellers:" +
                       station.getTotalNrOfPassengers());
    System.out.println("Total nr of persons transported in this simulation:" +
                       calcTotalNrOfPassengers(taxis));
  }

  /**
   * Calculates the total time of the simulation by looping over all taxis
   *
   * @param taxis
   * @return total time
   */
  private static int calcTotalTime(Taxi[] taxis) {
    int time = 0;
    for (Taxi taxi : taxis) {
      time = time + taxi.calcTotalTime();
    }
    return time;
  }

  /**
   * Calculates the total number of passengers that has been transported by
   * looping over all taxis
   *
   * @param taxis
   * @return total number of passengers
   */
  private static int calcTotalNrOfPassengers(Taxi[] taxis) {
    int total = 0;
    for (Taxi taxi : taxis) {
      total += taxi.getTotalNrOfPassengers();
    }
    return total;
  }

}
```

Method `step` handles the actual simulation: if there are passengers waiting at the station, the next taxi transports a number of persons according to its capacity or less, depending on the available number of persons at the station. If there are no passengers left, the next train can bring new passengers. The last train closes the station.

A possible output reads:

```
New taxi 1 created
New taxi 2 created
New taxi 3 created
New taxi 4 created
Train with 63 passengers has arrived
Taxi 1 takes 4 passengers
Taxi 2 takes 4 passengers
Taxi 3 takes 7 passengers
Taxi 4 takes 7 passengers
..
..
..
Taxi 1 takes 4 passengers
Taxi 2 takes 4 passengers
Taxi 3 takes 7 passengers
Taxi 4 takes 3 passengers
All persons have been transported
Total time of this simulation:350
Total nr of train travellers:744
Total nr of persons transported in this simulation:744
```

Method `takePassengers()` of class `Taxi` tries to take as many passengers as possible (limited by its capacity). The code reads:

```
/**
 * Tries to take maxNrOfPassengers from the station.
 * If actual number is less than that number is taken
 */
    public void takePassengers() {
        int passengersWaiting = station.getNrOfPassengersWaiting();
        if ( passengersWaiting > 0 ) {
            int nrOfPassengers = Math.min(passengersWaiting, maxNrOfPassengers);
            station.leaveStation(nrOfPassengers);
            totalNrOfPassengers += nrOfPassengers;
            nrOfRides++;
            System.out.println("Taxi " + taxiId + " takes " + nrOfPassengers +
                                                                    " passengers");
        } else {
            System.out.println("Taxi " + taxiId + " takes no passengers");
            try {
              // if no passengers at the station wait some time
              TimeUnit.MILLISECONDS.sleep(SLEEPTIME);
            }
            catch (InterruptedException e) {
              e.printStackTrace();
            }
        }
    }
```

The complete code is available in the file `taxirides.txt` at BlackBoard.

Assignment

Use the classes of the sequential version as the basis for a new concurrent version.

2 Stopping Threads

Although each `Thread` has a method `stop`, this is not the way to stop a thread. See the lecture and `https://docs.oracle.com/javase/8/docs/technotes/guides/ concurrency/threadPrimitiveDeprecation.html` for more details. The proper way to stop a thread in a safe way is that the `run` method checks every now and then whether it is supposed to stop. When this is the case, it should stop in such a way that all other threads interacting with this thread are not harmed. A simple way to indicate that a thread has to stop is by setting a boolean attribute of the task.

Assignment

Make a JavaFX program with an interface as depicted in Figure 2. When the `start` button is pressed the JavaFX program starts computing the Ackermann function with the arguments specified in the interface.

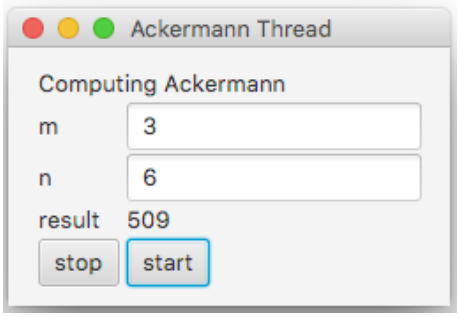

Figure 2: Interface of the Ackermann program.

The Ackermann function is defined as:

$$\text{Ackermann}(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ \text{Ackermann}(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ \text{Ackermann}(m-1, \text{Ackermann}(m,n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

Since the computation of such an Ackermann number can take very much time, you should use a separate thread for the computation in order to keep your program responsive.

Only the JavaFX thread is allowed to change the nodes in the interface. An attempt of any other thread to change this will cause an exception. Hence, the task computing the Ackermann number cannot set the result. The thread executing the event handler cannot wait until the computation is finished, this would make your program unresponsive. The proper solution is that the computation thread asks the JavaFX thread to update the result field. JavaFX provides `Platform.runLater(Runnable runnable)` for this purpose. It will run the specified `Runnable` on the JavaFX Application Thread at some unspecified time in the future.

Pressing the `stop` button should stop the running computation of the Ackermann number. Ensure that there appears a message in the interface indicating that the computation is interrupted.

The computation of an Ackermann number can consume huge amounts of stack space due to very deep nested recursive calls. When the stack space is exhausted the thread will throw an exception. In order to handle such an exception, you use the method

```
setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)
```

of the thread. The interface `UncaughtExceptionHandler` just contains the method

```
void uncaughtException(Thread t, Throwable e);
```

For example:

```
Thread t = new Thread(runnable);
t.setUncaughtExceptionHandler((Thread t, Throwable e) → System.out.println("Oeps"));
t.start();
```

Ensure that your Ackermann program gives a descent message in the primary stage telling that there was an exception.

3 Using `Worker` and `Task`

Since it is very common that one wants to stop a thread or to observe its progress, Java provides a standard solution in the form of the `Worker` interface and the classes `Task` and `Service`, both implementing `Worker`.

Most Tasks require some parameters in order to do useful work. Because Tasks operate on a background thread, care must be taken that external method calls do not modify the state during the computation. There are two techniques most useful for doing this: final variables and passing variables to a Task in the constructor.

The `Interface Worker`

The `Worker` interface defines an object that executes on a background thread. Each `Worker` has read-only properties observable from the JavaFX Application thread: title (String), message (String), running (Boolean), state (Enum Worker.State), totalWork (double), workDone (double), progress (double), value (Object), and exception (Object). The enumeration type `Worker.State` has states: READY, SCHEDULED, RUNNING, SUCCEEDED, CANCELLED, and FAILED. JavaFX provides two abstract classes implementing this interface: `Task<V>` and `Service<V>`. Services are intended for repeating workers, and are not considered here.

The **abstract** `Class Task<V>`

The **abstract** `Class Task<V>` is meant for *one-shot workers*. The state of `Task` objects is defined by the enumeration type `Worker.State` and can be depicted as in Figure 3.

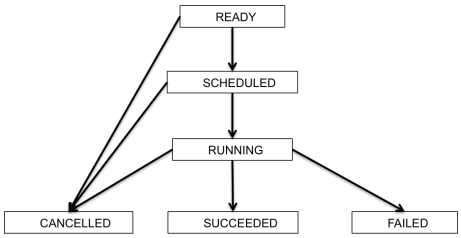

Figure 3: State transition diagram of a Task.

The class `Task<V>` implements the `Worker` interface and adds many methods like:

```
protected abstract V call() throws Exception
public final boolean cancel()
public boolean isCancelled()
protected void updateTitle(String title)
protected void updateMessage(String message)
protected void updateProgress(long workDone, long totalWork)
protected void updateValue(V value)
protected void succeeded()
protected void cancelled()
V getValue()
```

void `setOnScheduled(EventHandler<WorkerStateEvent> value)`
void `setOnSucceeded(EventHandler<WorkerStateEvent> value)`
void `setOnCancelled(EventHandler<WorkerStateEvent> value)`
void `setOnFailed(EventHandler<WorkerStateEvent> value)`

A Task can be executed by a thread similar to a runnable. The advantage of a Task is that it can be cancelled and that its progress can be determined. Note that a `Task` has an abstract method `V call()` **throws** `Exception` instead of the **void** `run()` of the `Runnable` interface. Hence, you should define `call` instead of `run`.

Assignment

Use the class `Task` and its native methods `cancel()` and `isCancelled()` to implement the interruptible Ackermann computation from the previous section. Use `setOnSucceeded` with an appropriate handler, or override `succeeded()` to update the GUI.

Deadline

Sunday May 29, 23:59.