Laboratory Practices
The ants problem
(2 sessions)

Concurrency and Distributed Systems

Introduction

The main goal of this practice is to analyze and complete a concurrent program that contains different synchronization conditions, using some of the Java tools provided by the *java.util.concurrent* library. When you complete the proposed tasks, you will have learned:

- How to correctly use the *ReentrantLock* tool.
- How to generate and use *Conditions* associated with a particular *ReentrantLock*.
- Apply solutions to the deadlock problem.

This practice consists of two sessions. You might also need additional time to complete the practice. In this document you will see some activities to be done. You should solve them and write the results, to facilitate further study of the content of the practice.

The ants problem

There is a territory where a set of ants live. The territory is modeled as a rectangular NxN matrix, where N is a program parameter. Ants are implemented as a Java threads, with a random initial position within the territory.

Each ant moves freely through the territory, moving at any moment to a cell adjacent to the current one (i.e., up, down, left or right). The initial number of ants can be specified, as well as the number of movements that each ant will make before finishing. When an ant finishes, it disappears from the territory (freeing the cell it was occupying).

Ants are modeled by the Ant class, which extends Thread. The basic code of the ants is:

```
class Ant extends Thread {
   ...
   public void run() {
      ...
      t.hi(a); // t = terrain. put ant a in a random cell
      while (movs > 0) {
         movs--;    // decrement remaining moves
         t.move(a); // move ant a to random next cell
         delay();   // applies a random delay
      }
      t.bye(a);     // ant a disappears
      ...
   }
}
```

As a restriction to the movements of ants, there can be a maximum of one ant in each territory cell:

- Initially, the program places each ant in a free cell.
- If an ant wants to move to an occupied cell, it will have to wait until the cell is freed.

The problem is solved using the ***monitor*** concept. When an ant wants to move to a cell and that cell is occupied, the ant must be suspended on a condition variable until it is reactivated by another ant. The Terrain interface specifies the operations of the monitor:

```
interface Terrain {
   void hi  (int a);
   void bye (int a);
   void move(int a) throws InterruptedException;
}
```

The methods *hi*, *bye* and *move* are invoked from the code of each ant. The simulation finishes when one of these conditions holds:

1. All the ants have finished their execution and have abandoned the territory.
2. A deadlock situation has been reached, and the remaining ants will never be able to finish.

In order to model the territory, different classes implementing Terrain are considered:

- **Terrain0**.- Basic monitor (implicit lock and condition).
- **Terrain1**.- General monitor (java.util.concurrent) with a single condition variable for the whole territory.
- **Terrain2**.- General monitor with one condition variable for each territory cell: an ant is suspended on the condition variable associated to the occupied cell it wants to move to.
- **Terrain3**.- General monitor with one condition variable for each territory cell. It also includes a mechanism to solve the deadlock problem.

The class Terrain0 is already implemented and must be used as the starting point for the other types of territories. The following sections presents the code provided. The rest of the sections of this document correspond to the different activities proposed, one for each type of territory.

Software provided

You can download the required code for the practice from the PoliformaT site (file Ants.jar).

The class Ants contains the main method. From a terminal, you can write:

```
java Ants
```

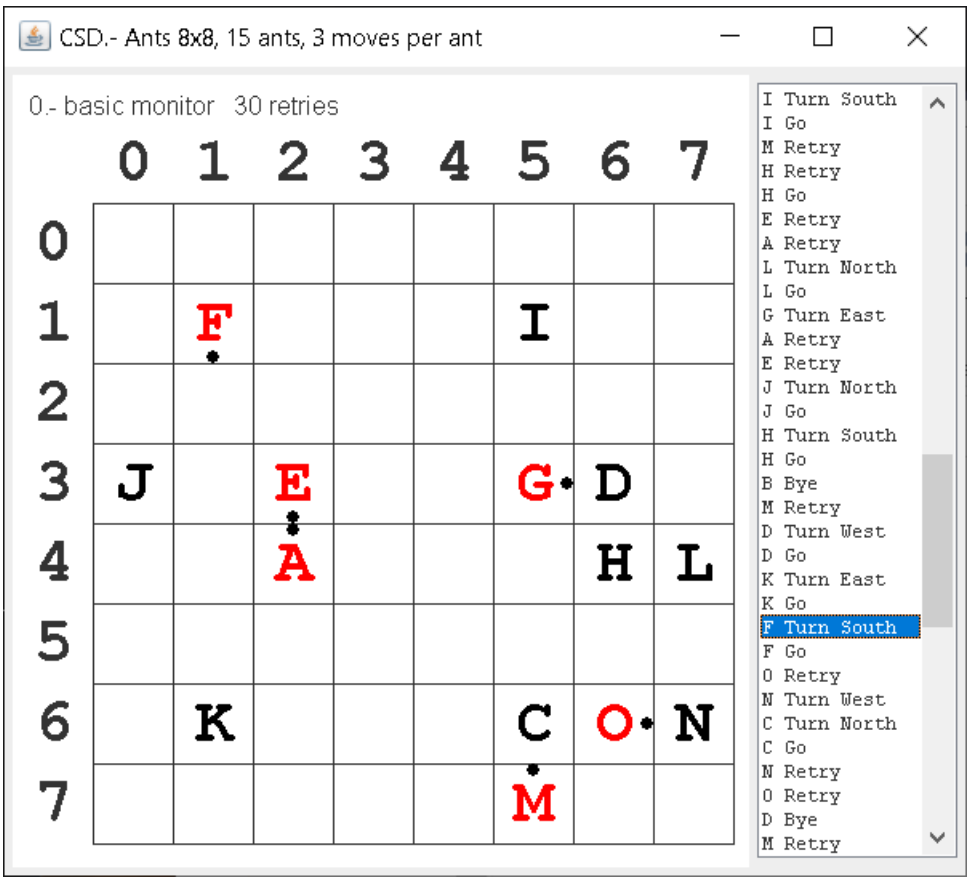
possibly followed by optional arguments. The acceptable arguments are:

- Type of Territory (0,1,2,3, with 0 as default value).
- Territory size (integer between 6 and 10, with 8 as default value).
- Number of ants (integer between 10 and 26, with 15 as default value).
- Number of movements per ant (integer between 2 and 5, with 3 as default value).

Examples:

- `java Ants` (Terrain0, 8x8 cells, 15 ants, 3 movements).
- `java Ants 1 10 26 2` (Terrain1, 10x10 cells, 26 ants, 2 movements).

When you the application is run, a window will be displayed with the territory and an event list. When any event from the list is selected, the corresponding state is shown in the territory. The following figure shows how the application will look like after selecting an event (`F Turn South`):



The keys to interpret the figure are:

- The grid on the left represents the state of the territory (initially empty), and the list on the right is the sequence of events. The contents of the territory represent the state at the point selected on the list of events (initially none).
- When an event of the list is selected, the territory state is shown. Events can be selected freely (next, previous, jump to a particular state, …).
- If an ant wants to move to another cell, it will be colored in red, with a dot indicating the direction it wants to move. Every ant that is not moving is colored in black.
- The following notation is used for the event list.

Event	**Meaning**
`X hi (r,c)`	Ant X arrives at the territory, and it is placed in the position (r=row, c=column).
`X turn dir`	Ant X wants to move to the adjacent cell in the direction dir (North, West, South, East).
`X go`	Ant X completes the movement (to the corresponding adjacent cell).
`X retry`	Ant X was waiting (occupied destination) and retries the movement.
`X chgDir`	Ant X was waiting (occupied destination) and has decided to change direction.
`X bye`	Ant X abandons the territory (the cell becomes empty).

The previous figure showing the program window can be interpreted in the following way:

- Ants not trying to move: I, J, D, H, L, K, C, N.
- Ants trying to move to a free destination: F
  - When the destination is free, the next event will correspond to the movement itself (event “F go” in the figure).
  - There can only be a maximum of one ant in this situation.
- Ants trying to move to an occupied destination: E, G, A, O, M.
  - E and A are in a deadlock situation: they will never be able to complete their movements.
  - The rest of the ants waiting (G, O, M) will be able to complete their movements when the destination cells become free.
- The window title indicates the simulation parameters.
- A message is shown at the top indicating the monitor type and the number of retries (*retry* events) carried out by all the ants during the whole simulation.

Code analysis

The provided code implements different classes. None of them must be modified:

- *Pos, Op, Hi, Bye, Turn, Retry, Go, ChgDir, Viewer* are “opaque classes", necessary for the application but without interest for the student.
- *Ant* is the class that extends *Thread* and implements the concept of ant. You need to understand how the method *run* (already presented) works.
- Interface *Terrain* (already commented).
- *Terrain0* is the starting point to implement the rest of the territories. You should analyze it to understand how it works. The operations on *v* (attribute of type *Viewer*) are necessary for the program to work correctly: they must be kept when developing *Terrain1*, *Terrain2* and *Terrain3*.

```
class Terrain0 implements Terrain {
   Viewer v;
   public Terrain0 (int t, int ants, int movs, String msg) {
      v=new Viewer(t,ants,movs,msg);
   }
   public synchronized void hi (int a) {v.hi(a); }
   public synchronized void bye (int a) { notifyAll(); v.bye(a);}
   public synchronized void move (int a)
          throws InterruptedException {
      v.turn(a); Pos dest=v.dest(a);
      while (v.occupied(dest)) {
         wait();
         v.retry(a);
      }
      v.go(a);
      notifyAll();
   }
}
```

- Code of classes Terrain1, Terrain2 and Terrain3 is not provided (they must be developed in this practice).
- *Ants* is the **main class**. It gets the arguments from the command line and launches the simulations using the different types of territory.

Activity 0 (Basic synchronization in Java)

Launch several executions of the provided code using *Terrain0* and the default values (*java Ants*) and complete the following table, which indicates for each execution:

- The total number of retries performed by the ants Many
- and the number of ants that are in a deadlock at the end. Sometimes only 2, other times all of them

Run	Total Retries	Number of deadlocked ants
1		
2		
3		
4		
5		
6		

Repeat the executions with the command *java Ants 0 8 10 3*

Run	Total Retries	Number of deadlocked ants
1		
2		
3		
4		
5		
6		

Still many retries and deadlocked ants, but not as many as with the default parameters

Activity 1 (Synchronization with ReentrantLocks in Java)

Implement *Terrain1* using the tools *ReentrantLock* and *Condition*, provided by the *java.util.concurrent library*.

Fill in the following tables:

java Ants 1

Run	Total Retries	Number of deadlocked ants
1		
2		
3		
4		
5		
6		

java Ants 1 8 10 3

Run	Total Retries	Number of deadlocked ants
1		
2		
3		
4		
5		
6		

Do you see significant differences between *Terrain0* and *Terrain1*?
Are there less "unnecessary" reactivations of threads in *Terrain1* compared to *Terrain0*?

There doesn't seem to be a significant difference in the unnecessary reactivations, retry() calls, or deadlocked ants.

Activity 2 (Using multiple Condition variables)

Implement Terrain2, using an array of conditions (as many conditions as cells).
NOTE.- The sentence *v.getPos(a)* returns the current position of ant a.
NOTE.- Given a position (e.g. *Pos dest=v.dest(a);*) we can obtain its *x* and *y* coordinates (*dest.x* and *dest.y*)

Fill in the following tables:
java Ants 2

Run	Total Retries	Number of deadlocked ants
1		
2		
3		
4		
5		
6		

java Ants 2 8 10 3

Run	Total Retries	Number of deadlocked ants
1		
2		
3		
4		
5		
6		

Do you see significant differences between *Terrain0*, *Terrain1* and *Terrain2*?

Activity 3 (Management of deadlocks)

Develop *Terrain3*, which will use one condition for each cell (like *Terrain2*), but it should also cope with the deadlock problem. When an ant has been waiting for a given maximum time (e.g. 300 ms) without being able to complete its movement, it changes direction by calling *v.chgDir(A)* (which recomputes the destination).
NOTE.- *await(long timeout, TimeUnit unit)* blocks until the corresponding *signal* (in which case it returns *true*) or until the indicated timeout has expired (in which case it returns *false*).

How would you classify the mechanism used to cope with the deadlock problem (prevention, avoidance, detection+recovery)? Which Coffman condition, if any, is broken?