IS1220 - Object Oriented Software Design

Tutorial 09

Paolo Ballarini

General instructions:

- If you have not done so already, create a workspace for your tutorial `IS1220/TDs`.
- Within the workspace, create a new package for this TD called `fr.ecp.IS1220.TD09/`.
- Carefully document your code (JavaDoc), i.e., explain the general idea of your algorithm, explain the relevant steps in the code implementing the algorithm, document assumptions (if any), corner cases, and error conditions.

Learning outcomes:

- Multi-threading programming
- Observer Pattern

Exercise 1. First simple example of multi-threaded program

Import file `Tutorial9_ex1_2complete.zip` in Eclipse. It contains a simple example of a Java multi-threaded program, consisting of classes: `PrintDemo`, `ThreadDemo` and `TestThreadDemo`. Specifically `PrintDemo` defines a `printCount()` method which simply displays a sequence of messages on the screen (each message displays a numeric value indicating the order of the message being displayed). `ThreadDemo` implements a Thread through the `run` in which he prints the message `printCount()` through a `PrintDemo` attribute. Finally `TestThreadDemo` creates two `ThreadDemo` *threads* and runs them until completion.

```
class PrintDemo {
   public void printCount(){
    try {
         for(int i = 5; i > 0; i--)
            System.out.println("Counter ---- " + i );
     } catch (Exception e) {
         System.out.println("Thread interrupted.");
     }
   }

}

class ThreadDemo extends Thread {
   private String threadName;
   private PrintDemo PD;

   ThreadDemo( String name, PrintDemo pd){
       threadName = name;
        PD = pd;
   }
   public void run() {
     PD.printCount(threadName);
     System.out.println("Thread " + threadName + " exiting.");
   }
}

public class TestThreadDemo {
   public static void main(String args[]) {
      PrintDemo PD = new PrintDemo();
      ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD );
      ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD );

      T1.start();
      T2.start();

      // wait for threads to end
      try {
         T1.join();
         T2.join();
      } catch( Exception e) {
         System.out.println("Interrupted");
      }
   }
}
```

Tasks.

- run the `main` method of class `TestThreadDemo` different times and observe the output displayed on the console: is the output of each execution consistent ? explain
- based on class `ThreadDemo` define a modified class called `ThreadDemoSynchronized` so that the message displayed by different threads are guaranteed not to interleave (what means you have to ensure *thread-safeness* ?)
- define a second version of the above code in which threads are created through implementation of the `Runnable` interface, rather than through subclassing of the `Thread` class (as it is the case with the `ThreadDemo` class).

Exercise 2. Concurrent Bank Account

Two brothers want to share a bank account and need to have access to it for performing basic operations (reading the balance, making a deposit, withdrawing money) for example through internet banking. A possible JAVA implementation of a program for managing the shared account is based on the definition of classes `Account` (providing operations `getBalance`, `deposit` and `withdraw`) and `AccountHolder` (representing the activity of each one of the brothers holding the account), which , in a first attempt, are defined as follows:

```
class Account {
        private double balance;

        public Account(double initialDeposit) {
            balance = initialDeposit;
        }

        public double getBalance() {
            return balance;
        }

        public void deposit(double amount) {
            balance += amount;
       }

        public void withdraw(double amount) {
            if ( balance >= amount ) { balance -= amount; }
       } // no negative balance allowed

   }

class AccountHolder extends Thread {
        private Account acc;
```

```
    public AccountHolder(Account a) {
        acc = a;
    }

    public void run() {
        ...
        acc.withdraw(100);
        acc.deposit(200);

        ...
    }
}
```

what are the possible problems with this implementation? is the withdraw operation guaranteed to yield the correct (desired) behavior? and the deposit operation ? If not explain why.

Task. Modify the above implementation (class `Account` and `AccountHolder`) so to avoid thread interference. Write a small test program displaying the trace of execution (displaying the balance before and after each withdrawal and deposit operation) of the `AccountHolder` threads so to check that the program behaves correctly. In order to make the result of different interleaving you may want to insert some pause (i.e. `Thread.sleep()`) in the execution of the "withdraw" and "deposit" operations. If you do so repeated executions of you test program should display different inter leavings.

Exercise 3. Observing basketball matches

A sport internet web-site provides its registered used with the possibility to follow in real-time the evolution of basketball matches. Once logged in into the system users can access a table of ongoing games and and select one game they want to follow (i.e. to observe). Such system must support multiple users concurrently (i.e. several users can be connected to the system at observe concurrently the ongoing games).

Task. Based on the Observer design pattern define a JAVA program for representing the above described system. The specifications of the basic elements of such program are the following ones:

- Game class: stores the score of both teams; stores the name of match; stores a string representation of the current score of the game; keeps track of the identity of the team who holds the ball; scoring points: a teams score can change only when the team holds the ball, in this case the score is increased by a random (integer) value in the set 0,1,2,3.
- Viewer class (representing observer users): stores the identity of the user; stores the

identity of the observed game; stores the string representation of the current score of the observed game.

- To exploit the Observer pattern you can (if you want) take advantage of the Observer interface (java.util.Observer) and Observable class (java.util.Observable).

To test your program write a Test class where you initialise at least 3 matches, and 3 users which register to different matches. Let the matches start and run for a finite (sufficiently long duration) and display the result observed by the 3 users. To double check your system try also a configuration where at least 2 users observe the same match.