Laboratory 5

Threads of execution in Java SE – *java.util.concurrent* package

- Part 1 -

1. Laboratory objectives

- presentation of the reserved word *volatile*;
- presentation, understanding and use of the following aspects of the *java.util.concurrent* package:
    - *thread-safe* collections,
    - *Lock, ReentrantLock,*
    - Semaphore class.

2. Theoretical considerations, Example

The java.util.concurrent package was introduced in JDK version 5.0.

2.1 The reserved word volatile

The *volatile* keyword is used to show that a variable can be modified by multiple threads. This variable will be in the *main memory*. Using the keyword *volatile* reduces the risk of memory consistency errors: if a *volatile* variable changes, this change is visible to all threads. Using these atomic variables is more efficient than accessing the variables by synchronized methods, but requires more attention from the programmer to avoid errors in memory consistency [MAN, 2008].

Example:

```
volatile int primitive; //declaration of a volatile attribute
volatile String reference;
```

2.2 *thread-safe* Collection

A code sequence is *thread-safe* if it works with common data in a way that can guarantee data security when it is accessed by multiple threads of execution.

The class hierarchy (Framework) that implements the *Collection* interface (introduced in *JDK 1.2*) is very flexible, benefiting from the three basic models (interfaces) List, Set and Map. Some of the *framework* classes are already *thread-safe* (*Hashtable* and *Vector*), and for others there are encapsulation modes (*wrappers*).

Example:

```
// insecure collections
List unsafeList = new ArrayList();
Set unsafeSet = new HashSet();
Map unsafeMap = new HashMap();

// sercure collections
List threadSafeList = Collections.synchronizedList(new ArrayList());
Set threadSafeSet = Collections.synchronizedSet(new HashSet());
Map theadSafeMap = Collections.synchronizedMap(new HashMap());
```

Therefore, if a collection can be accessed by more than one thread of execution, it needs to be *thread-safe*, either by encapsulating it through a *Factory* class that implements synchronization mechanisms, or by choosing a secure implementation (which is already *thread-safe*).

All collection iterators in the *java.util* package fail if the list is modified (by adding or deleting elements) during iteration, throwing an exception of the *ConcurrentModificationException* type.

This issue has been solved (for lists and sets) by introducing two new implementations: *CopyOnWriteArrayList* and *CopyOnWriteArraySet* (from JDK 5.0). These classes create a copy of the collections (list, set respectively) iterating each time an element is added or deleted.

Concerning *Map* collections, the *ConcurrentHashMap* class was introduced, with a much better performance than the older versions (*Hashtable* or *Collections.synchronizedMap*(*new HashMap* ())). *ConcurrentHashMap* allows reading, writing, and deleting at the same time without locking the collection (unlike previous secure variants).

The iterators introduced in the *java.util.concurrent* package do not ensure data consistency during the iteration, meaning that it may or may not "notice" deleted or added objects (during iteration), but this is a preferred failure (throwing the *ConcurrentModificationException* exception). However, in general, browsing a collection is much more frequent than adding and deleting elements [GOE, 2008].

The non-blocking queues (described by the *Queue* interface) are implemented in Java through:

- *PriorityQueue* – it is not *thread-safe.*
- *ConcurrentLinkedQueue* – is *thread-safe,* rapid, FIFO.

Block queues (described by the *BlockingQueue* interface) cause a thread to write to a filled up list or read from an empty list (as opposed to non-blocking ones). The implementations of this list category are (all *thread-safe*):

- *LinkedBlockingQueue,*
- *PriorityBlockingQueue,*
- *ArrayBlockingQueue,*
- *SynchronousQueue,*
- *DelayQueue.*

2.3 *Lock* synchronization

Locks are obtained by implementing the *Lock* interface and work in a similar manner to the default monitors used by *synchronized* blocks. As with the default monitors, only one thread can acquire and hold one lock at a time. They also support the *wait*() and *notify*(), *notifyAll*() methods.

One of the main advantages of *Lock* objects is the ability to check the availability of a monitor before trying to acquire it, by calling the *tryLock*() method. If the lock is taken, an alternative logic can be executed.

This method also has a form with *tryLock* parameters (*long time*, *TimeUnit unit*) where the maximum waiting time for the lock release can be specified through them. If the thread is interrupted while trying to acquire the lock, an exception of type *InterruptedException* is thrown that must be handled.

The *lockInterruptibly*() method acquires the lock unless the thread is interrupted, in this case it throws an exception of type *interruptedException* [ORA, 2011].

The *newCondition*() method returns a *Condition* object associated with the lock. By calling the *await*() and *signal*() methods for the *Condition* object, a *wait*/*notify* mechanism can be implemented [ORA, 2012].

The main differences between <u>Locks</u> and synchronized blocks:

- synchronized blocks do not guarantee that the sequences where the threads are waiting to enter the block will receive access;
- the parameters cannot be set when entering the synchronized blocks, the access cannot be monitored;
- not having the support of the block programming (as in the case of *synchronized*), the responsibility of calling the *unlock*() method is the programmer's.

The *ReentrantLock* class is one of the implementations of the *Lock* interface and besides the methods in this interface are worth mentioning:

- *getOwner*() - returns the thread reference that holds the lock or *null*;
- *getQueuedThreads*() - returns a collection of threads that are probably waiting for the lock;
- *getQueueLength*() - returns the estimated number of threads that are waiting for the lock to be acquired;
- *getWaitingThreads*(*Condition condition*) - returns a collection containing those threads waiting to meet the *Condition* condition associated with this lock;
- *getWaitQueueLength*(*Condition condition*) - returns an estimated number of threads that can wait for meeting the *Condition* condition associated with this lock;
- *hasWaiters*(), returns a true (false) variable if there is (or not) at least one thread waiting for the lock to be acquired.

<u>Example:</u>

Requirements: To implement a simple application that exemplifies the basic functionality of the locks (Lock). The operating mode of the lock type synchronization mechanism is shown in the sequence diagram in Figure 5.1.

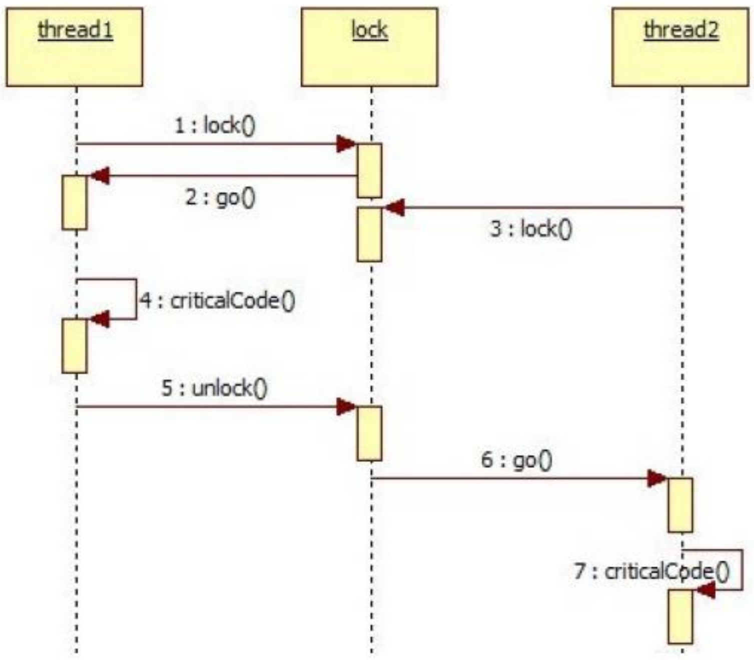

Figure 5.1 The sequence diagram for the locks example

It is worth mentioning that in fact, the *Lock* object does not call any *go*() method. The only purpose for which this method appears in the application is to present the mechanism in the most intuitive way.

The **implementation** of the application is presented in code sequence 1.

Code sequence 1: Example of implementing a mutual exclusion through locks

```
class Fir extends Thread {
      int name;
      Lock l;
      Fir(int n, Lock la) {
            this.name = n;
            this.l = la;
      }
      public void run() {
            this.l.lock();
            System.out.println("The thread " + name + " acquired the lock");
            criticalRegion ();
            this.l.unlock();
            System.out.println("The thread " + name + " released the lock");
      }
      public void criticalRegion() {
            System.out.println("IS EXECUTING THE CRITICAL REGION!");
            try {
                  Thread.sleep(3000);
            } catch (InterruptedException e) {
                  e.printStackTrace();
      }
}
}
public class Main {
      public static void main(String args[]) {
            Lock l = new ReentrantLock();
            Fir f1, f2;
            f1 = new Fir(1, l);
            f2 = new Fir(2, l);
            f1.start();
            f2.start();
      }
}
```

2.4 Semaphore Synchronization

In essence, a semaphore is an integer that can be incremented or decremented respectively by two or more processes through special functions. These functions also ensure that processes are blocked or unblocked when the semaphore reaches a certain limit.

The *Semaphore* class in *Java SE* has two basic methods: *acquire*() and *release*().

The *acquire*(*int permits*) method blocks the thread of execution that calls it until the internal value of the semaphore is at least equal to the number specified as a parameter of the method.

The *release*(*int permissions*) method increases the internal value of the semaphore by a value that is equal to the number specified as a parameter of the method.

The *aquire*() and *release*() methods are overloaded so that they have no parameters. Using this variant of the methods and initializing the semaphore with the value 1, a mechanism that is similar to the locks (binary semaphore) is obtained.

The constructor of this class is overloaded (*Semaphore*(*int permissions*) and *Semaphore*(*int permissions, boolean fair*)). When the *fair* parameter is true, the semaphore guarantees that the blocked threads of execution are given access in the order of the *acquire*() method, according to the *FIFO* model. In general, semaphores are used to control the access to resources to avoid starvation of threads of execution [ORA, 2010].

Example:

Requirements: To implement a *Java SE* application that models the *Petri* net in Figure 5.2.

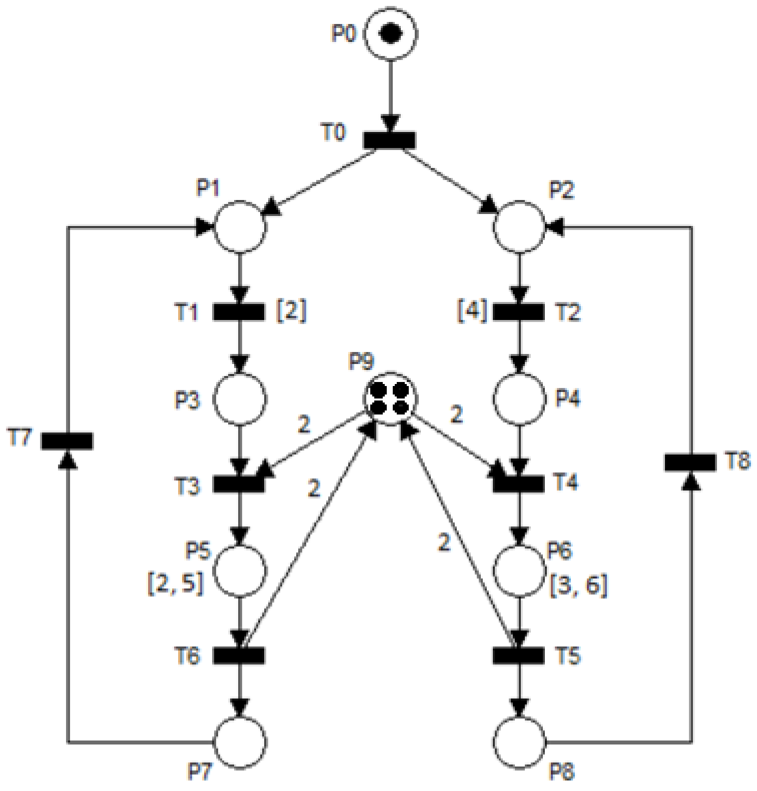

Figure 5.2 Example of semaphores application

Specifications:

The application will be implemented in *Java SE*, using a semaphore for the synchronization element at place *P9*.

The timings of places *P5* and *P6* correspond to some activities of the form:

```
int k = ...;//random no. within
      //a specified interval
for (int i = 0; i < k * 100000; i++) {
      i++;
      i--;
}
```

Timed transitions *T1* and *T2* represent delays and will be implemented by calling the *Thread.sleep*(*x*) method.

The **implementation** of the application is presented in Code Sequence 2.

Code sequence 2: Example for implementing a mutual exclusion using semaphores

```
import java.util.concurrent.Semaphore;

class Fir extends Thread {
	int name, delay, k, permit;
	Semaphore s;

	Fir(int n, Semaphore sa, int delay, int k, int permit) {
		this.name = n;
		this.s = sa;
		this.delay = delay;
		this.k = k;
		this.permit = permit;
	}

	public void run() {
		while (true) {
			try {
				System.out.println("Fir " + name +" State 1");
				Thread.sleep(this.delay * 500);
				System.out.println("Fir " + name +" State 2");
				this.s.acquire(this.permit); // critical region
				System.out.println("Fir " + name + " took a token from the
semaphore");
				System.out.println("Fir " + name +" State 3");
				for (int i = 0; i < k * 100000; i++) {
					i++;
					i--;
				}
				this.s.release(); // end of critical region
				System.out.println("Fir " + name + " released a token from the
semaphore");
				System.out.println("Fir " + name +" State 4");
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}
}

public class Main {
	public static void main(String args[]) {
		Semaphore s = new Semaphore(4);
		Fir f1, f2;
		f1 = new Fir(1, s, 2, (int) Math.round(Math.random() * 3 + 2), 2);
		f2 = new Fir(2, s, 4, (int) Math.round(Math.random() * 3 + 3), 2);
		f1.start();
		f2.start();
	}
}
```

3. Developments and tests

3.1 Application 1

The demonstration applications made available during the laboratory hours will be tested and understood.

3.2 Application 2

For the example of semaphore application (subchapter 2.4), the class diagram and the sequence diagram will be prepared.

3.3 Application 3:

Problem statement:

Implement the *producer-consumer* application, using a non *thread-safe* collection (for example *ArrayList*).

Specifications:

The application will consist of a *producer* thread and three *consumer* threads. The thread will generate random numbers that it will be stored in a non *thread-safe* collection for a period of one second. The *consumer* threads will extract these numbers and display them in the console (along with the current thread name).

The application will guarantee concurrent access to the collection by using lock type synchronizations.

It will guarantee the smooth operation of the application even if the collection is empty.

The collection capacity will be limited to 100 items.

Requirements:

The application design will be achieved using the *Petri* nets and the class diagram.

The application will be implemented in *Java SE*, using *Condition* locks and objects.

Test:

The application will be implemented so that it can test the synchronizations.

A locking mechanism (on request) of the *consumer* threads will be implemented, so that the condition regarding the collection capacity can be tested.

3.4 Application 4

Problem Statement:

To implement a simple simulator for the control of road traffic in an intersection like the one in Figure 5.3.

Specifications:

The controlled intersection has only one lane per direction, and at one point a single traffic light indicates green.

The simulator will consist of 5 threads of execution: a thread responsible for generating the cars for the 4 waiting queues (the number of cars generated will be a reasonable one). The other 4 threads of execution will try to acquire the traffic light and will allow extracting a maximum of 10 cars from the waiting queues.

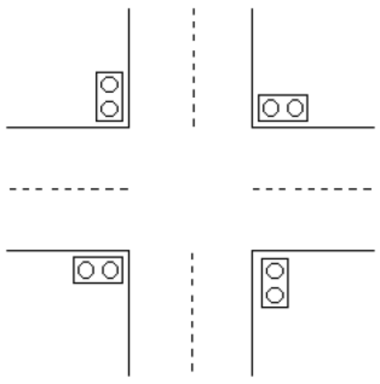

Figure 5.3 Application 4

The 4 traffic lights will be implemented using a single *Semaphore* object, as follows:

- when at least two cars appear in the queue, they will try to requisition all the 10 traffic light permits;
- the traffic light will guarantee the access of the threads of execution in FIFO order;
- the time required for a car to leave the intersection is one second; therefore, the traffic light will be requisitioned by a thread of execution for a number of seconds that are equal to the number of cars in the waiting queue (but maximum 10 seconds).

Requirements:

The application will be modeled by the class diagram and state machines. The application will be implemented in *Java SE*. The waiting queues will be modeled by simple numerical values (synchronized access must be guaranteed). The cars generation rates in the waiting queues will be configured in a text file. The application does not have to have a graphical interface. However, the presence of the graphical interface will a bonus.

Test:

If the application does not have a graphical interface, additional efforts will be made to design / implement it to demonstrate its functionality.

The logic functionality of the two types of threads of execution will be tested.

The synchronization between the threads of execution will be tested.

The access to the configuration text file will be tested.

4. Knowledge verification

1) Explain the purpose of the reserved word *volatile*.

2) What do you mean by the phrase: *thread-safe* collections?

3) What are the main differences between implementing mutual exclusion through synchronized blocks and locks?

4) What are the main differences between the implementation of mutual exclusion through locks and semaphores?

5) Explain how you would implement a similar mechanism to *wait/notify* using semaphores? What would be the benefits of such an implementation?