Laboratory 6

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

- Part 2 -

1. Laboratory objectives

- presentation, understanding and use of the following aspects of the *java.util.concurent* package:
  - a *CyclicBarrier* class,
  - a *CountDownLatch* class,
  - an *Exhanger* class.

2. Theoretical considerations

2.1 CyclicBarrier class

The *CyclicBarrier* class implements a synchronization mechanism that ensures that a group of threads of execuion are expecting each other at a common point (the *await*() method call). The execution of the threads will be suspended until all the threads of execution in the group call the method mentioned above.

The barrier is called *cyclic* because it is reusable; the barrier is reset to its original state when all the threads met at the barrier and were released. Also a time limit for waiting at the barrier can be specified; if during this time the rest of the threads have not reached the barrier, it is considered that the barrier has been broken and all waiting threads "at the barrier" will receive an exception from the *await*() method of the type *BrokenBarrierException* [GOE, 2008].

Constructors:

- *CyclicBarrier*(*int parts*) - creates a barrier that can synchronize a number of threads equal to the integer parameter given to the constructor.
- *CyclicBarrier(int parts, Runnable barrierAction*) - creates a barrier that can synchronize a number of threads of execution equal to the integer parameter given to the constructor; In addition, it executes the routine of the *Runnable* object (implemented in the *run*() method), given as the second parameter, once all the threads have reached the barrier and before unlocking them; this routine will run on a newly created thread.

Methods:

- *int await*() - suspends the thread or threads until the threshold defined by the *CyclicBarrier* object is reached, then all threads continue execution together;
- *int await*(*long timeout, TimeUnit unit*) - suspends the thread or threads until the threshold defined by the *CyclicBarrier* object or for a time defined by the *timeout* parameter is reached. The *TimeUnit* type argument defines the time unit;
- *int getNumberWaiting*() - returns the number of threads waiting for the barrier;
- *int getParties*() - returns the parameter in the object's definition;
- *boolean isBroken*() - returns the *true* value if there are threads that have passed the barrier after an exception or because the waiting time has elapsed and *false* otherwise;
- *void reset*() - the barrier is reset [ORA, 2011a].

Example:

The sequence diagram in Figure 6.1 shows the operation of the *cyclic barrier* synchronization mechanism.

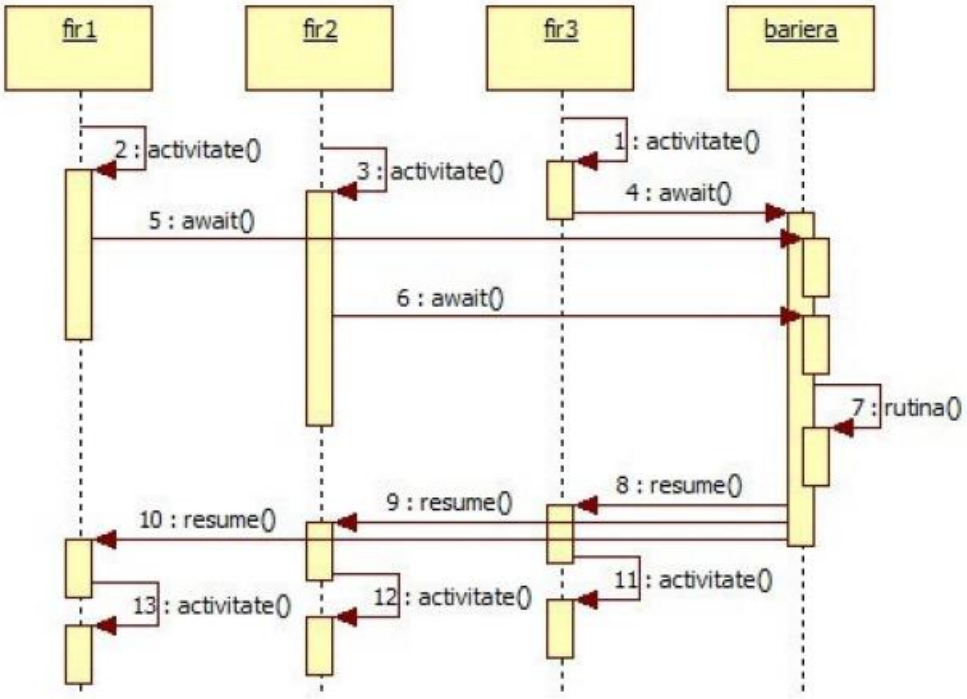

Figure 6.1 Operation of a cyclic barrier

The **implementation** of the application described by the previous sequence diagram is presented in Code Sequence 1.

Code sequence 1: cyclic barrier example

```
class Fir extends Thread {
	CyclicBarrier bariera;
	public Fir(CyclicBarrier bariera) {
		this.bariera = bariera;
	}
	public void run() {
		while (true) {
			activitate();
			try {
				bariera.await();
			} catch (InterruptedException e) {
				e.printStackTrace();
			} catch (BrokenBarrierException e) {
				e.printStackTrace();
			}
		activitate();
	}
}
	public void activitate() {
		System.out.println(this.getName() + "> activitate");
		try {
			Thread.sleep(Math.round(Math.random() * 3 + 3) * 1000);
		} catch (InterruptedException e) {
		}
	}
}
```

```
	public class Main {
		public static void main(String args[]) {
			CyclicBarrier bariera = new CyclicBarrier(3, new Runnable() {
				public void run() {
					System.out.println("Barrier Rutine");
				}
			});
			Fir fir1 = new Fir(bariera);
			Fir fir2 = new Fir(bariera);
			Fir fir3 = new Fir(bariera);
			fir1.start();
			fir2.start();
			fir3.start();
	}
}
```

2.2 CountDownLatch class

This class is similar to the *CyclicBarrier* class, having the role of coordinating the execution of a group of threads. The *CountDownLatch* class has a constructor with an integer argument (*int count*) that indicates the initial value of the counter, but unlike *CyclicBarrier* it is not reusable. Each thread can decrement the class counter by invoking the *countDown*() method. This method does not block the threads of execution that call it, but it only decrements the counter. The *await*() method acts a little differently, any thread that invokes this method is blocked until the counter reaches zero, at which point all the threads are released. There is also the *await*(*long timeout*, *TimeUnit unit*) variant, with a semantic similar to the *CyclicBarrier* class.

This class is useful when a problem can be broken down into several parts, each thread receiving a sub problem.

A *CountDownLatch* class with the counter set to value 1 can be used as a starting gate for a group of threads. The threads of execution wait for the "signal", and the main thread (coordinator) decrements the counter which releases all the threads at once [GOE, 2008].

Another difference from the *CyclicBarrier* class is the absence of the optional constructor routine [ORA, 2011b].

Example:

As an example for using the class, Code Sequence 2 is proposed.

Code sequence 2: Example for using *CoutDownLatch* class

```
public void run() {
	while (true) {
		activitate1();
		countDownLatch.countDown();
		try {
			countDownLatch.await();
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		activitate2();
	}
}
```

2.3 Exchanger class

The *Exchanger* class implements two communication channels between two cooperating threads. The way this class works is similar to that of a *CyclicBarrier* instance, with the counter set to 2; In addition, the *Exchanger* class allows the exchange of information between the two threads (which they use for synchronization), once they have reached the *rendezvous* point [GOE, 2008] [ORA, 2011c].

A typical use of this class is in problems where one thread fills one *buffer* and another consumes information (of the same type) from another *buffer*. When both threads are finished filling or emptying buffers, they are interchanged.

Example: code sequence 1 presents a simple example for using the *Exchanger* class.

Code sequence 3: Example for using the *Exchanger* class

```
class Fir extends Thread{
      List<Integer> list=new ArrayList<Integer>();
      int sleepTime;
      Exchanger<List<Integer>> exchanger;
      String name;

      Fir(int sT, Exchanger<List<Integer>> exchanger,String name){
            this.sleepTime=sT;
            this.exchanger=exchanger;
            this.name =name;
      }

      public void displayList(){
             //displays the list of the current thread
              for (int i = 0; i < this.list.size(); i++) {
                   System.out.println(this.list.get(i));
              }

      }

      public void run(){
            int noElem=(int)Math.round(Math.random()*3+1);
            for(int i=0;i<noElem;i++){// populate the list with a random
                                     // number of items
                  int elem=(int)Math.round(Math.random()*100);
                  list.add(new Integer(elem));
            }
            this.displayList();//display the list before exchange
            try {
                  Thread.sleep(this.sleepTime);// the thread is waiting x ms
                                               // the exchange of objects is made
                  this.list=exchanger.exchange(this.list);
            } catch (InterruptedException e) {
                  e.printStackTrace();
            }
            this.displayList();//display the list after the exchange
      }
}

public class ExchangerTestSimple {
      public static void main(String args[]){
            Exchanger<List<Integer>> exchanger =
                  new Exchanger<List<Integer>>();
            Fir f1=new Fir(1000,exchanger,"Duke",1);
            Fir f2=new Fir(5000,exchanger,"Wild Wings",1);
            f1.start(); f2.start();
      }
}
```

3. Developments and tests

3.1 Application 1

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

3.2 Application 2

Problem statement:

Design and implement an application consisting of three threads of execution. The application will simulate solving an iterative mathematical problem. Thus, in each iteration, the three threads of execution will calculate a random integer between -10 and 10. The three results from an iteration will be summed together and written in a text file along with the iteration number. When the sum of the partial results is equal to 0, the number of iterations performed by each thread will be displayed.

Requirements:

Design: class diagram and Petri net.

Implementation: Java SE, using the *CyclicBarrier* class

Testing:

Will test:

- synchronization mechanism - the number of iterations for all 3 threads should be the same;
- Writing correctly in the text file.

3.3 Application 3

Problem statement:

Design and implement an application consisting of two threads of execution. One of the threads is waiting for text messages in the console. After reading a number of 10 messages, this thread passes the messages to another thread that also displays them in the console, reversing the order of the characters in the message. Once the cycle is completed, the application restarts the routine.

Requirements:

Design: class diagram and state machines.

Implementation: Java SE, using the *Exchanger* class.

Testing:

will be tested:

- the synchronization mechanism;
- the correct functionality of the two threads of execution (independently).

3.4 Application 4

Problem statement:

Imagine, design and deploy an application that exemplifies the operating mechanism of the *CountDownLatch* class.

Requirements:

Design: Choose the tools you find most useful.

The application will be implemented in *Java SE*.

4. Knowledge verification

1) Explain the purpose and operation of the *CyclicBarrier* synchronization mechanism?
2) Explain the purpose and operation of the *CountDownLatch* synchronization mechanism?
3) Highlight the most important differences between *CyclicBarrier* and *CountDownLatch* synchronization mechanisms.
4) Explain the purpose and operation of the *Exchanger* synchronization mechanism?