TD 1 - Java concurrency: synchronizers

To set up for the practical exercises go to CELENE and download `code.zip`. Create a Java project in your favorite IDE (e.g., Eclipse, Netbeans) and import the contents of `code.zip` into your project. You should now have 6 packages, one per exercise, called `polytech.tours.di.parallel.td1.exo#` where `#` is the number of the exercise. You're now good to go.

1 Thread interference

The objective of this first exercise is to see thread interference in action. Study the three classes in package `polytech.tours.di.parallel.td1.exo1`, namely, `Counter`, `ParrallelCounting`, and `Tester`. Run the `main()` method of class `Tester` several times. Do you observe any abnormal behavior? if so, can you explain it?

2 Synchronize methods and sections

Refactor the code in package `polytech.tours.di.parallel.td1.exo2` so the thread interference is avoided.

Hint: remember the synchronize methods and synchronize sections we discussed in class.

3 Explicit locks

We saw in class that `java.util.concurrent.ReentrantLock` provides a ready-to-go implementation of a re-entrant lock for thread synchronization. For learning purposes we will reinvent the wheel and code our own implementation (but please in real applications use the one provided by Java).

Code a class called `polytech.tours.di.parallel.td1.exo3.Lock` implementing two methods `lock()` and `unlock()`. The `lock()` method locks the `Lock` instance so that all threads calling `lock()` are blocked until `unlock()` is executed.

Hint: remember the `wait()` and `notify()` methods we studied in class.

To test your `Lock`, refactor class `polytech.tours.di.parallel.td1.exo3.Counter` so it uses an instance of your `Lock` to prevent memory inconsistency errors and thread interference. You can use `polytech.tours.di.parallel.td1.exo3.Tester` to conduct the experiments.

Hint: we saw an example of these guarded blocks in class.

Listing 1: Solution: Lock.java

```java
package polytech.tours.di.parallel.td1.exo3;

/**
 * Implements a lock
 * @author Jorge E. Mendoza (dev@jorge-mendoza.com)
 *
 */
public class Lock {
```

```java
        /**
         * True if the lock is locked and false otherwise
         */
        private boolean isLocked = false;
        /**
         * Acquires the lock.
         *
         * @throws InterruptedException
         */
        public synchronized void lock() throws InterruptedException{
                while(isLocked){
                        System.out.println(Thread.currentThread().getName() +
                                        " waiting for the lock @ "+ System.nanoTime());
                        wait();
                }
                isLocked = true;
                System.out.println(Thread.currentThread().getName() +
                                " obtained the lock @ "+ System.nanoTime());
        }
        /**
         * Releases the lock
         */
        public synchronized void unlock(){
                System.out.println(Thread.currentThread().getName() +
                                " released the lock @ "+ System.nanoTime());
                isLocked = false;
                notify();
        }
}
```

Listing 2: Solution: Counter.java

```java
package polytech.tours.di.parallel.td1.exo3;

import java.util.ConcurrentModificationException;

/**
 * Implements a simple counter
 * @author Jorge E. Mendoza (dev@jorge-mendoza.com)
 * @version %I%, %G%
 *
 */
public class Counter{

        //the lock
        private Lock lock;

        //class constructor
        public Counter(){
                this.lock=new Lock();
        }

        /**
         * The count
```

```java
	 */
	private int count=0;

	/**
	 * Increments the counter
	 */
	public void inc(){
		try{
			lock.lock();
			count++;
		}catch (InterruptedException e){
			throw new ConcurrentModificationException();
		}finally{
			lock.unlock();
		}
	}
	/**
	 * Decrements the counter
	 */
	public void dec(){
		try{
			lock.lock();
			count--;
		}catch (InterruptedException e){
			throw new ConcurrentModificationException();
		}finally{
			lock.unlock();
		}
	}
	/**
	 *
	 * @return the count
	 */
	public int getCount(){
		return this.count;
	}

}
```

4 Is your `Lock` re-entrant?

Study class the `polytech.tours.di.parallel.td1.exo4.ReentrantTask`. Implement a class called `polytech.tours.di.parallel.td1.exo4.Tester` with a `main()` method that launches the execution of an instance of `ReentrantTask` in a `Thread`. What happens? why?

Listing 3: Solution: Tester.java

```java
package polytech.tours.di.parallel.td1.exo4;

public class Tester {

	/**
	 *
	 * @param args[0] the number of counter increments that each task should perform
	 */
	public static void main(String[] args){
		(new Thread(new ReentrantTask())).start();
```

```
        }

}
```

5 Making our Lock re-entrant

Implement a class called `polytech.tours.di.parallel.td1.exo5.ReentrantLock` that solves the reentrance problem. Use `polytech.tours.di.parallel.td1.exo5.Tester` to test your solution.

Hint: remember that a thread may try to obtain the same lock more than twice.
Hint: remember than a thread executing a task is nothing but an instance of class `Thread`.

Listing 4: Solution: ReentrantLock.java

```
package polytech.tours.di.parallel.td1.exo5;

/**
 * Implements a simple re-entrant lock
 * @author Jorge E. Mendoza (dev@jorge-mendoza.com)
 * @version %I%, %G%
 *
 */
public class ReentrantLock {
        /**
         * True if the lock is locked and false otherwise
         */
        private boolean isLocked = false;
        /**
         * A reference to the thread holding the lock
         */
        private Thread lockedBy;
        /**
         * The number of times the thread has acquired the lock
         */
        int lockedCount = 0;

        /**
         * Acquires the lock.
         *
         * @throws InterruptedException
         */
        public synchronized void lock() throws InterruptedException{
                //get a reference to the calling thread
                Thread callingThread = Thread.currentThread();
                while(isLocked && callingThread!=lockedBy){
                        System.out.println(Thread.currentThread().getName()
 +
                                        " waiting for the lock @ "+ System.
    nanoTime());
                        wait();
                }
                isLocked = true;
                lockedBy=callingThread;
                lockedCount++;
                System.out.println(Thread.currentThread().getName() +
                                " obtained the lock @ "+ System.nanoTime())
    ;
```

```
		}
		/**
		 * Releases the lock
		 */
		public synchronized void unlock(){
			if(Thread.currentThread()==lockedBy){
				lockedCount--;
				if(lockedCount==0){
					System.out.println(Thread.currentThread().getName() +
							" released the lock @ "+System.nanoTime());
					lockedBy=null;
					isLocked = false;
					notify();
				}
			}
		}
}
```

6 Implementing a cyclic barrier

A cyclic barrier is a synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. Cyclic barriers are useful in programs involving a fixed sized party of threads that must occasionally wait for each other. The barrier is called cyclic because it can be re-used after the waiting threads are released (for further details check page 72 of the class slides for chapter 3). Package `java.util.concurrent` in the Java high level concurrency API contains a class implementing a cyclic barrier (`java.util.concurrent.CyclicBarrier`). The objective of this exercise is to implement our own (reduced) version of that class.

1. Study the sample usage for a cyclic barrier reported in the javadoc for class `java.util.concurrent.CyclicBarrier` (available at: `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CyclicBarrier.html`)

2. Study interface `CyclicBarrier` in package `polytech.tours.di.parallel.td1.exo6`

3. Develop a class called `polytech.tours.di.parallel.td1.exo6.MyCyclicBarrier` providing a concrete implementation of the interface

4. Test your cyclic barrier using class `polytech.tours.di.parallel.td1.exo6.Tester`

Listing 5: Solution: ReentrantLock.java

```
package polytech.tours.di.parallel.td1.exo6;

/**
 * Simple (and incomplete) implementation of a cyclic barrier
 * @author Jorge E. Mendoza (dev@jorge-mendoza.com)
 * @version %I%, %G%
 *
 */
public class MyCyclicBarrier implements CyclicBarrier {

		/** The total number of parties (threads) */
		private final int parties;
		/** The number of parties yet to arrive */
		private int count;
		/** The action to execute when the barrier is tripped */
```

```java
	private Runnable barrierAction;

	//EXO 1: class constructor
	public MyCyclicBarrier(int parties, Runnable barrierAction){
		if (parties <= 0) throw new IllegalArgumentException(); // Bonus +1
		this.parties=parties;
		this.count=parties;
		this.barrierAction=barrierAction;
	}

	//EXO 2: await method
	/* (non-Javadoc)
	 * @see U0111.CyclickBarrier#await()
	 */
	@Override
	public synchronized void await() throws InterruptedException{

		//decrements awaiting parties by 1.
		count--;
		//If the current thread is not the last to arrive, thread will wait.
		if(count>0){
			wait();
		}
		/*If the current thread is last to arrive,
		 *notify all waiting threads, and run the action*/
		else{
			/* All parties have arrive, make reset the counter
			 * so that MyCyclicBarrier could become cyclic. */
			count=parties;
			barrierAction.run(); //run barrier action
			notifyAll(); //notify all waiting threads

		}

	}

	//EXO4: getNumberWaiting
	/* (non-Javadoc)
	 * @see U0111.CyclickBarrier#getNumberWaiting()
	 */
	@Override
	public synchronized int getNumberWaiting(){
		return parties - count;
	}

	//EXO4: getParties();
	/* (non-Javadoc)
	 * @see U0111.CyclickBarrier#getParties()
	 */
	@Override
	public int getParties(){
		return this.parties;
	}

}
```

TD 2 - Java concurrency: solving liveness problems

To set up for the practical exercises go to CELENE and download `code.zip`. Create a Java project in your favorite IDE (e.g., Eclipse, Netbeans) and import the contents of `code.zip` into your project. You should now have 3 packages, one per exercise, called `polytech.tours.di.parallel.td2.exo#` where `#` is the number of the exercise. You are now good to go.

1 Experimenting a deadlock

The objective of this exercise is to analyze code leading to a deadlock. Study classes `polytech.tours.di.parallel.tp2.exo1.ParallelTask` and `polytech.tours.di.parallel.tp2.exo1.Tester`. Run the `exo1.Tester.#main` method. What do you observe? Can you explain the situation?

To help analyzing the problem you can take a look at the java thread dump of the application. Launch the Java Visual Virtual Machine (`jvisualvm`) while running your code and execute a dump thread. If you do not know how to lauch `jvsualvm`, check the documentation provided by Oracle at `https://docs.oracle.com/javase/8/docs/technotes/guides/visualvm/intro.html`.

2 Solving deadlocks

We are now going to try to avoid the deadlock situation using different strategies.

2.1 Lock ordering

If you did your homework (of course you did), you read about lock ordering as one of the most common strategies to avoid deadlocks. Refactor class `polytech.tours.di.parallel.tp2.exo2_1.Tester` so it makes sure that the three threads always access the locks in the same order. Run some experiments with your refactored class. What do you observe?

Listing 1: Solution — Counter.java

```java
package polytech.tours.di.parallel.td2.exo2_1;

import java.util.concurrent.locks.ReentrantLock;

/**
 * Runs parallel tasks
 * @author Jorge E. Mendoza (dev@jorge-mendoza.com)
 * @version %I%, %G%
 *
 */
public class Tester {

	/**
	 * Runs the test
	 * @param args no arguments expected
	 * @throws InterruptedException
	 */
	public static void main(String[] args) throws InterruptedException {

		ReentrantLock lock1=new ReentrantLock();
		ReentrantLock lock2=new ReentrantLock();
```

```java
                ReentrantLock lock3=new ReentrantLock();

                Thread t1 = new Thread(new ParallelTask(lock1,lock2), "t1");
                Thread t2 = new Thread(new ParallelTask(lock1,lock2), "t2");
                Thread t3 = new Thread(new ParallelTask(lock1,lock2), "t3");

                t1.start();
                Thread.sleep(5000);
                t2.start();
                Thread.sleep(5000);
                t3.start();

        }
}
```

2.2 Avoiding nested locks

Nested locks is probably the most common reason for deadlocks. Therefore, avoiding locking one resource if you already hold one is always a good idea. Refactor class `polytech.tours.di.parallel.tp2.exo2_2.ParallelTask` to avoid nested locks. Run method `exo2_2.Tester.#main`. What do you observe?

Listing 2: Solution — Lock.java

```java
package polytech.tours.di.parallel.td2.exo2_2;

import java.util.concurrent.locks.ReentrantLock;

/**
 * Implements a parallel task that needs two locks to execute
 * @author Jorge E. Mendoza (dev@jorge-mendoza.com)
 * @version %I%, %G%
 *
 */
class ParallelTask implements Runnable{

        /**
         * A reference to the first lock (resource)
         */
        private ReentrantLock lock1;
        /**
         * A reference to the second lock (resource)
         */
        private ReentrantLock lock2;
        /**
         * Constructs a new instance of the class
         * @param lock1 a reference to the first lock
         * @param lock2 a reference to the second lock
         */
        public ParallelTask(ReentrantLock lock1, ReentrantLock lock2){
                this.lock1=lock1;
                this.lock2=lock2;
        }
        /**
         * The code of the parallel task
         */
        @Override
        public void run() {
                String callingThread = Thread.currentThread().getName();
                System.out.println(callingThread+" acquiring "+lock1);
                lock1.lock();
                try{
                        System.out.println(callingThread+" acquired "+lock1);
                        work();
```

```
                    }finally{
                            System.out.println(callingThread+" released "+lock1);
                            lock1.unlock();
                    }
                    System.out.println(callingThread+" acquiring " + lock2);
                    lock2.lock();
                    try{
                            System.out.println(callingThread+" acquired " + lock2);
                            work();
                    }finally{
                            lock2.unlock();
                    }
                    System.out.println(callingThread+" released " + lock1);
                    System.out.println(callingThread+" finished execution.");
            }

            /**
             * Simulates an expensive computation
             */
            private void work() {
                    try {
                            Thread.sleep(30000);
                    } catch (InterruptedException e) {
                            e.printStackTrace();
                    }
            }
}
```

2.3 Lock timeout

Another deadlock prevention mechanism is to put a timeout on lock attempts. Under this mechanism a thread trying to obtain a lock will only try for a given (and usually pre-defined) time before giving up. If a thread does not succeed in taking all necessary locks within the given timeout, it will backup, free all locks taken, then retry or abandon its task.

Refactor class `polytech.tours.di.parallel.tp2.exo2_3.ParallelTask` so the threads wait for 5 seconds on the locks. If a thread cannot obtain a lock, it should free all acquired locks and abort its execution. Run method `polytech.tours.di.parallel.tp2.exo2_3.Tester` to test your implementation.

Hint: class `ReentrantLock` provides a method signed `public boolean tryLock(long timeout, TimeUnit unit)` that allows a thread to acquire the lock if it is not held by another thread within the given waiting time and the calling thread has not been interrupted. Check the Javadoc.

Listing 3: Solution — Lock.java

```
package polytech.tours.di.parallel.td2.exo2_3;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantLock;

/**
 * Implements a parallel task that needs two locks to execute
 * @author Jorge E. Mendoza (dev@jorge-mendoza.com)
 * @version %I%, %G%
 *
 */
class ParallelTask implements Runnable{

        /**
         * A reference to the first lock (resource)
         */
        private ReentrantLock lock1;
```

```
	/**
	 * A reference to the second lock (resource)
	 */
	private ReentrantLock lock2;
	/**
	 * Constructs a new instance of the class
	 * @param lock1 a reference to the first lock
	 * @param lock2 a reference to the second lock
	 */
	public ParallelTask(ReentrantLock lock1, ReentrantLock lock2){
		this.lock1=lock1;
		this.lock2=lock2;
	}
	/**
	 * The code of the parallel task
	 */
	@Override
	public void run() {
		String caller = Thread.currentThread().getName();
		System.out.println(caller+" acquiring "+lock1);
		try{
			if(lock1.tryLock(5,TimeUnit.SECONDS)){
				System.out.println(caller+" acquired "+lock1);
				work();
				System.out.println(caller+" acquiring "+lock2);

				if(lock2.tryLock(5,TimeUnit.SECONDS)){
					System.out.println(caller + " acquired "+lock2);
					work();

				}else{
					System.out.println(caller+" could not acquire "+lock2);
					lock1.unlock();
					System.out.println(caller+" released "+lock1);
					System.out.println(caller+" aborting execution");
					return;
				}
			}
			else{
				System.out.println(caller+" could not acquire "+lock1);
				System.out.println(caller+" aborting execution");
				return;
			}
		} catch (InterruptedException e) {
			return;
		}finally{
			if(lock1.isHeldByCurrentThread()){
				lock1.unlock();
				System.out.println(caller+ " released "+lock1);
			}
			if(lock2.isHeldByCurrentThread()){
				lock2.unlock();
				System.out.println(caller+ " released "+lock2);
			}
		}
		System.out.println(caller+" finished execution.");
	}

```

```
        /**
         * Simulates an expensive computation
         */
        private void work() {
                try {
                        Thread.sleep(30000);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
}
```

3 Fairness

Study the classes in package `polytech.tours.di.parallel.tp2.exo3_1`. Run several times method `polytech.tours.di.parallel.tp2.exo3_1.Tester#main` with different values for T and D. What do you observe?

3.1 Implementing fairness 1

Class `ReentrantLock` includes a constructor that accepts a fairness parameter. When set true, under contention, the method favors granting access to the longest-waiting thread. Study the Javadoc for this constructor and refactor class `polytech.tours.di.parallel.tp2.exo3_1.Testers#main` so it implements fairness. Repeat your experiments. What do you observe?

3.2 Implementing fairness 2

For teaching purposes, and also to have some fun, we are going to implement our own version of `ReentrantLock` providing fairness. Implement methods `lock()` and `unlock()` in class `polytech.tours.di.parallel.tp2.exo3_2.FairLock`. Run method `polytech.tours.di.parallel.tp2.exo3_2.Tester#main`. Do you obtain results that are similar to those obtained in exercise 3.1?.

Hint: Use objects of class `polytech.tours.di.parallel.tp2.exo3_2.QueueObect` to build a queue of objects on which calling threads can wait on.

Listing 4: QueueObject.java

```
package polytech.tours.di.parallel.td2.exo3_2;

public class QueueObject {
        /**
         * Stores a signal
         */
        private boolean isNotified = false;
        /**
         * Checks if a signal has been sent to the calling thread. If that is the case
         * the method completes its execution, otherwise it waits on this object.
         * makes the calling thread wait on this object.
         * @throws InterruptedException
         */
        public synchronized void doWait() throws InterruptedException {
                while(!isNotified){
                        this.wait();
                }
                this.isNotified = false;
        }
        /**
         * Sets the signal on and notifies the thread waiting on this object.
         */
        public synchronized void doNotify() {
```

```java
                this.isNotified = true;
                this.notify();
        }

        @Override
        public boolean equals(Object o) {
                return this == o;
        }
}
```

Listing 5: Solution — Lock.java

```java
package polytech.tours.di.parallel.td2.exo3_2;

import java.util.ArrayList;
import java.util.List;

/**
 * Our own version of a reentrant lock providing fairness
 * @author Jorge E. Mendoza (dev@jorge-mendoza.com)
 * @version %I%, %G%
 *
 */
public class FairLock {
        /**
         * True if the lock is locked and false otherwise
         */
        private boolean isLocked = false;
        /**
         * A reference to the thread holding the lock
         */
        private Thread lockingThread  = null;
        /**
         * A list of objects on which waiting threads wait
         */
        private List<QueueObject> queue = new ArrayList<QueueObject>();
        /**
         * Locks the lock
         * @throws InterruptedException
         */
        public void lock() throws InterruptedException{
                //Creates an queue object representing the calling thread's position in the queue
                QueueObject queueObject = new QueueObject();
                //Assume the lock is locked for the calling thread
                boolean lockedByAnotherThread = true;
                synchronized(this){
                        //Get in the queue
                        queue.add(queueObject);
                }
                //While the lock is locked by another thread
                while(lockedByAnotherThread){
                        synchronized(this){
                                /*if the lock is still locked or the calling thread is
                                 *not the fist in the queue
                                 */
                                lockedByAnotherThread=isLocked||queue.get(0)!=queueObject;
                                if(!lockedByAnotherThread){
                                        isLocked = true;
                                        queue.remove(queueObject);
                                        lockingThread = Thread.currentThread();
```

```
                    return; //The calling thread obtained the lock
                }
            }
            //If the calling thread cannot acquire the lock then wait
            try{
                queueObject.doWait();
            }catch(InterruptedException e){
                synchronized(this) {
                    queue.remove(queueObject);
                }
                throw e;
            }
        }
    }
    /**
     * Unlocks the lock
     */
    public synchronized void unlock(){
        if(this.lockingThread != Thread.currentThread()){
            throw new IllegalMonitorStateException("Calling thread has not locked this lock");
        }
        isLocked = false;
        lockingThread = null;
        if(queue.size() > 0){
            queue.get(0).doNotify();
        }
    }
}
```

TD 3 - Java concurrency: parallel algorithms

To set up for the practical exercises open your favorite IDE (e.g., Eclipse, Netbeans) and create a Java project. You're now good to go.

1 Estimating the value of π

The value of π can be calculated in a number of ways. Consider the approach below:

- Inscribe a circle in a 1x1 square
- Randomly generate points in the square
- Determine the number of points in the square that are also in the circle
- Let p be the number of points in the circle divided by the number of points in the square, then $\pi \approx 4 \times p$

Figure ?? illustrates the approach.Note that the quality of the approximation increases with the number of generated points.

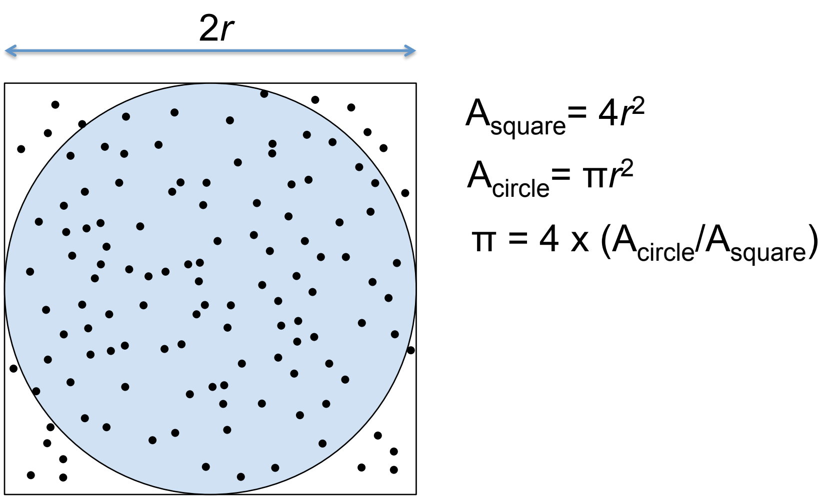

Figure 1: Approximating π

It is possible to solve this problem using what is known as an embarrassingly parallel solution; that is a solution which is computationally intensive and has minimal communication and minimal I/O. The objective of this exercise is to come up with such solution.

1. Propose a parallel algorithm for this problem
   - What decomposition strategy better fits the problem?
   - What is the task dependency graph of your algorithm?
   - What is the task interaction graph of your algorithm?
2. Propose a Java implementation for your algorithm

Listing 1: Solution — ScoreComputation.java

```java
package polytech.tours.di.parallel.td3.pi;

import java.util.concurrent.Callable;
import java.util.concurrent.ThreadLocalRandom;

/**
 * Implements a parallel tasks that estimates PI using simulation
 * @author Jorge E. Mendoza (dev@jorge-mendoza.com)
 * @version %I%, %G%
 *
 */
public class PIComputation implements Callable<Long> {
	//replications
	private long runs;
	//circle diameter
	private int d;
	//the in-the-circle points counter
	private long count;
	//the x coordinate of the circle/square center
	private double ox;
	//the y coordinate of the circle/square center
	private double oy;
	/**
	 * Constructs a new PIComputation
	 * @param runs the "size" of the task, that is, the number of runs (simulated points) for which the task is responsible.
	 * @param d the diameter of the circle (or the side of the square)
	 */
	public PIComputation(long runs, int d){
		this.runs=runs;
		this.d=d;
		ox=d/2d;
		oy=d/2d;
	}

	@Override
	public Long call() throws Exception {
		count=0l;
		for(int i=1; i<=runs; i++){
			double px=ThreadLocalRandom.current().nextDouble(d);
			double py=ThreadLocalRandom.current().nextDouble(d);
			if(inCircle(px,py))
				count++;
		}
		return count;
	}
	/**
	 *
	 * @param cx the x coordinate of a point
	 * @param cy the y coordinate of a point
	 * @return true if the point <code>(cx,cy)</code> is insie the circle, false otherwise
	 */
	private boolean inCircle(double cx, double cy){
```

```java
                return euclidean(cx,cy,ox,oy)<=d/2d;
        }
        /**
         *
         * @param cx1 the x coordinate of the first point
         * @param cy1 the y coordinate of the first point
         * @param cx2 the x coordinate of the second point
         * @param cy2 the y coordinate of the second point
         * @return the Euclidean distance between point <code>(cx1,cy1)</code> and point <code>(cx2,cy2)</code>
         */
        private double euclidean(double cx1, double cy1, double cx2, double cy2){
                return Math.sqrt(Math.pow(cx1-cx2, 2)+Math.pow(cy1-cy2, 2));
        }

}
```

Listing 2: Solution — ScoreComputation.java

```java
package polytech.tours.di.parallel.td3.pi;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
/**
 * Provides services for estimating the value of PI using monte-carlo simulation
 * @author Jorge E. Mendoza (dev@jorge-mendoza.com)
 * @version %I%, %G%
 *
 */
public class PICalculator {
        /**
         * Estimates the value of PI using monte-carlo simulation and parallel computing. The approach follows:
         * <ul>
         * <li>Inscribe a circle in a 1x1 square</li>
         * <li>Randomly generate <code>runs</code> points in the square</li>
         * <li>Determine the number of points in the square that are also in the circle</li>
         * <li>Let <code>p</code> be the number of points in the circle divided by the number
         * of points in the square, then <code>\pi \approx 4 \times p</code>
         *
         * The problem is decomposed into tasks as follows. Each task consists on generating  a given number of points
         * and evaluating if they lay (or not) in the circle. The number of points that each task
         * generates is called the task size. The number of tasks is a parameter.
```

```
	 *
	 * Tasks are executed in multiple threads. The task to thread mapping strategy is dynamic and managed
	 * by a {@link ExecutorService}. The number of available threads is a parameter.
	 *
	 * @param runs the number of points to simulate
	 * @param nbThreads the number of threads to use
	 * @param nbTasks the number of tasks. Each task has a "size" of <code>runs/nbTasks</code>
	 * @return an estimation of PI
	 */
	public double computePI(long runs, int nbThreads, int nbTasks){

		//Instantiate a service executor
		ExecutorService executor = Executors.newFixedThreadPool(nbThreads);
		//Initialize auxiliary variables
		List<Future<Long>> results;
		List<Callable<Long>> tasks=new ArrayList<Callable<Long>>();
		//Create the nbTasks and store them on a list
		for(int t=1; t<=nbTasks; t++){
			tasks.add(new PIComputation(runs/nbTasks,1));
		}
		try {
			//Ask the executor to execute the tasks and store the results on a list
			results=executor.invokeAll(tasks);
			executor.shutdown();
		} catch (InterruptedException e) {
			return Double.NaN;
		}
		//Compute PI
		try {
			long counter=0l;
			for(Future<Long> t:results){
				counter=counter+t.get();
			}
			return 4d * ((double)counter/runs);
		}
		catch (InterruptedException | ExecutionException e) {
			return Double.NaN;
		}
	}

}
```

Listing 3: Solution — ScoreComputation.java

```
package polytech.tours.di.parallel.td3.pi;
/**
 * Tests PI calculator
 * @author Jorge E. Mendoza (dev@jorge-mendoza.com)
 * @version %I%, %G%
 *
 */
public class Tester {
	/**
```

```
     * Runs experiments on {@link PICalculator} using different parameter values
     */
    public static void main(String[] args) {

        //experimental set up
        long runs[] = {1_000, 10_000, 100_000, 1_000_000, 10_000_000, 100_000_000, 1_000_000_000};
        int tasks[]={1, 10, 100, 1_000};
        int threads[]={1, 10, 20, 50, 100};
        for(int i=1; i<=100_000; i++); //warm up

        //run experiment for each combination of parameter values
        System.out.println("RUNS\tTASKS\tTHREADS\tCPU\tPI");
        for(int r=0;r<runs.length;r++){
            for(int t=0; t<tasks.length; t++){
                for(int p=0; p<threads.length; p++){
                    //experiment
                    PICalculator calculator=new PICalculator();
                    long start=System.currentTimeMillis();
                    double pi=calculator.computePI(runs[r], threads[p], tasks[t]);
                    long end=System.currentTimeMillis();
                    System.out.println(runs[r]+"\t"+tasks[t]+"\t"+threads[p]+"\t"+(end-start)/1000d+"\t"+pi);
                }
            }
        }
    }

}
```