The objective of this assignment is to understand the functionality provided by conditional variables and practice using the wait(), notify(), and notifyAll() methods. In addition, it allows for exercising how to interrupt threads.

Exercise 1 (Acquire the expertise) – Woodworker (Railway Quest)

This exercise has to be solved using *Railway Quest*.

Part A

Solve the first level of the third lesson in Railway Quest, titled “Level 6 - Woodworker”. Open the source code 'Level6WoodWorker.java' in your IDE and analyze it in detail. The code initializes two trains, registers them for the quest, and moves them along their respective tracks. Additionally, it initializes a *Crane* instance, which is needed to transfer goods between the trains.

The red train is responsible for picking up wood from the deposit using the *load()* method, while the green train unloads the wood at the factory using the *unload()* method. For the crane to successfully transfer wood between the trains, both the red and green trains must be present at the crane at the same time. Only one train needs to call the *crane.transfer()* method—it does not matter which one. However, once the transfer begins, both trains must wait until the transfer is fully completed before leaving the crane area.

The goal of the exercise is to ensure that all wood is successfully transported from the deposit to the factory.

Start by running the program to observe its current behaviour, then modify the code as needed to properly perform load/unload operations and to coordinate the trains' actions at the crane.

Part B

For the synchronization part of this exercise, implement an alternative solution using intrinsic locks. Use the *wait()* and *notify()/notifyAll()* methods to manage the sequence of train operations, ensuring that the wood is transferred and transported efficiently.

Exercise 2 (Problem solving) – Palio di Siena

Develop a program that simulates the departure of 10 jockeys at the “Palio di Siena”. All jockeys arrive at different times at the starting line (randomly generated between 1000 and 1500 ms – use the *ThreadLocalRandom* class) and must wait for the last jockey to arrive. The race begins only when the last jockey arrives at the starting line. Each jockey must measure the waiting time (the time elapsed from his arrival at the starting line until the race begins) and print it to the console.

Use conditional variables in combination with explicit locks.

A possible sample output:

```
Jockey2: reached starting line
Jockey8: reached starting line
Jockey7: reached starting line
Jockey5: reached starting line
Jockey0: reached starting line
Jockey9: reached starting line
Jockey4: reached starting line
Jockey3: reached starting line
```

```
Jockey1: reached starting line
Jockey6: reached starting line
Jockey6: waited 0 ms
Jockey2: waited 39 ms
Jockey7: waited 30 ms
Jockey0: waited 21 ms
Jockey8: waited 38 ms
Jockey9: waited 21 ms
Jockey5: waited 26 ms
Jockey4: waited 21 ms
Jockey1: waited 8 ms
Jockey3: waited 7 ms
```

Exercise 3 (Acquire the expertise) – Monte Carlo

Create a multithreaded program to approximate the value of π using the Monte Carlo method.

Five threads continuously generate random 2D points (x, y) within [0, 1], then check if the point is inside the unit circle and finally update two counters: the total number of generated points and the number of points that fall inside the unit circle.

The main thread, on the other hand, is responsible to initialize and start the five threads. Then, every 250 milliseconds, it computes, using the two counters updated by all other threads, the estimated value of π and its precision (using the Math.PI constant as reference). The main thread then prints to the console the current approximation and the precision. Upon reaching a precision of 6 digits or after exceeding 20 seconds of simulation, the main thread stops all the other threads using the interrupt mechanism, printing to the console whether the precision or the timeout has been reached.

To implement the Monte Carlo method, use the following snippets:

```
// performed by compute threads
generate random point (x, y) within [0, 1] x [0, 1]
totalPoints++
if (x^2 + y^2) <= 1
    insideCircle++

// performed by main thread to compute PI
pi_estimate = 4 * (insideCircle / totalPoints)
```

A possible sample output:

```
Compute Thread 0 starting ...
Compute Thread 1 starting ...
Compute Thread 2 starting ...
Compute Thread 3 starting ...
Compute Thread 4 starting ...
0.25s computedPi: 3.1413718590 (precision: 2.21E-04)
0.50s computedPi: 3.1414923489 (precision: 1.00E-04)
0.75s computedPi: 3.1415905806 (precision: 2.07E-06)
Precision reached: interrupting threads.
Compute Thread 2 terminating.
Compute Thread 3 terminating.
Compute Thread 0 terminating.
Compute Thread 4 terminating.
Compute Thread 1 terminating.
```

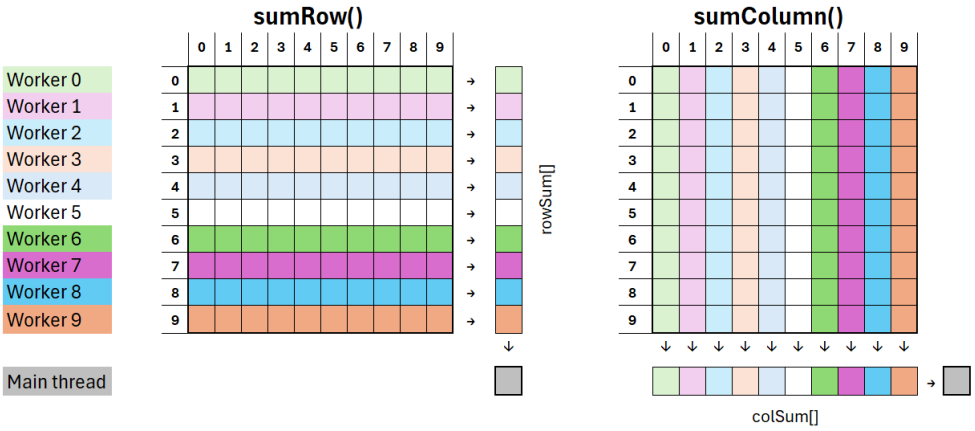
Exercise 4 (Optional) – Sum of rows and columns of a matrix

Open the source code 'MatrixRowColSums.java' and analyze it in detail. The program performs simple computations on a matrix (sum of rows and sum of columns).

Currently, the program operates sequentially. The objective of the exercise is to transform it into a multi-threaded version to perform computations concurrently.

The main thread is responsible to initialize a matrix using random numbers, create and start worker threads that are designated to compute the sum of each row and column. It must then wait until all the sums of the rows have been computed before it can proceed to calculate and display the total sum of these rows. Similarly, it needs to ensure that all the sums of the columns are completed before it computes and displays the total sum of the columns. At the end, the main thread has to wait for the termination of all the worker threads.

Each worker thread must compute the sum of values within a given row of the matrix. Once a worker thread completes its calculation, it must wait until all other worker threads have finished their row computations before starting to compute the sum of values within a given column of the matrix. When the sum of the column is computed, the worker thread can terminate.



Implement two versions of the program using:

- Conditional variables with explicit locks
- wait() and notify()/notifyAll() with implicit locks