Exercise 1

The program suffers from memory visibility problems on the shared variable *isRunning*. When the program runs on a computer with at least two cores, the worker threads remain blocked in the "while (!isRunning) {}" loop. Without the activation of a memory barrier, threads are unable to correctly see the variable's modifications and continue reading the old value. Since the variable is only modified by the main thread, while all the other threads only read its value, this problem can be solved by declaring the variable as volatile.

After this modification, if the program is executed again, it cannot terminate because of an additional memory visibility problem related to the *finished* variable. This variable is read by the main thread and is increased by each worker thread when their execution ends. Changing this variable to a volatile variable could seem to solve the visibility problem. Unfortunately, the atomicity of the increment operation would not be guaranteed (the finished++ operation is a read, increment, and write!). This means that two or more threads might manage to finish at the same time, read the same value, perform the increment operation and write the same result, therefore missing one increment during the operation. To guarantee the atomicity of the *increment operation,* it is possible to enclose the operation in a synchronized block or introduce synchronized methods that operate on the *finished* variable. An easier solution is to transform the variable from *int* to *AtomicInteger,* replace the *increment* operation with an *incrementAndGet() method* call and modify the reading of the variable with the *get() method.*

```
class Worker implements Runnable {
    public static volatile boolean isRunning = false;
    public static AtomicInteger finished = new AtomicInteger(0);

    private int count = 0;
    private final int id;
    private final Random random;

    public Worker(final int id) {
        this.id = id;
        this.random = new Random();
    }

    @Override
    public void run() {
        System.out.println("Worker" + id + " waiting to start");
        while (!isRunning) {
            // Wait!
        }

        System.out.println("Worker" + id + " started");
        for (int i = 0; i < 10; i++) {
            count += random.nextInt(40) + 10;
            try {
                Thread.sleep(random.nextInt(151) + 100);
            } catch (final InterruptedException e) {
                e.printStackTrace();
            }
        }

        System.out.println("Worker" + id + " finished");
        finished.incrementAndGet();
    }

    public void printResult() {
        System.out.println("Worker" + id + " reached " + count);
    }
}

public class A3Exercise1 {
    public static void main(final String[] args) {
```

```
        final List<Worker> allWorkers = new ArrayList<>();
        final List<Thread> allThread = new ArrayList<>();
        for (int i = 1; i <= 10; i++) {
            final Worker target = new Worker(i);
            allWorkers.add(target);
            final Thread e = new Thread(target);
            allThread.add(e);
            e.start();
        }

        try {
            Thread.sleep(1000);
        } catch (final InterruptedException e) {
                e.printStackTrace();
        }

        System.out.println("Main thread starting the race!");
        Worker.isRunning = true;

        while (Worker.finished.get() < allWorkers.size()) {
            // Wait
        }

        for (final Worker worker: allWorkers) {
            worker.printResult();
        }

        for (final Thread thread: allThread) {
            try {
                thread.join();
            } catch (final InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Exercise 2

The program has a memory visibility problem: all changes made by the main thread on shared variables (*home*, *office, mobile*, *emergency*, *version*) might not be correctly visible by the *Contact* threads.

The proposed solution completely avoids using implicit or explicit locks, exploiting the property of memory visibility extension of volatile variables. All variables visible from a thread before writing a volatile variable become visible to another thread that performs a reading of the same volatile variable.

In this case, since the main thread always writes the *home*, *office*, *mobile*, and *emergency* variables in the same order before the *version variable,* and all the *Contact* threads read the *version* variable as the first shared variable, it is possible to take advantage of the effect of extension of the memory visibility of volatile variables, by just declaring the *version variable* as volatile. As soon as the main thread changes the *version* variable, all threads will see both the new *version* variable value, as well as the updated *home*, *office*, *mobile*, and *emergency* variables. The same applies when substituting the volatile variable with an AtomicInteger variable.

```
public class A3Exercise2 {

    // Shared phone numbers of business man
    public static int home;
    public static int office;
    public static int mobile;
    public static int emergency;
    public static volatile int version;
...
}
```

Exercise 3

Initially, it is required to remove all code related to the explicit lock and transform the shared variable from *int* to *AtomicInteger*. Then, the code has to be refactored by introducing two local variables: *current* and *update*. The value of *current* is assigned with the initial value of the atomic variable *shared*, while the *update* variable will contain the new value that will be assigned to *shared* using the compareAndSet method of the AtomicInteger class. Since the CAS operation can fail, all the operations must be performed within a do-while loop.

In the original version, only one thread at a time can access the locked code area, while all other threads wait for the lock to be released. The CAS version, on the other hand, allows multiple players to remark that the shared counter has exceeded the maximum threshold and attempt to set the value to 0. For this reason, it is required to reassign the variable *isRunning* (responsible for ending the loop) inside the do-while loop. Only the thread that succeeds in the CAS operation can terminate while all others will have to restart the operation.

```
class Player implements Runnable {
    private static final AtomicInteger shared = new AtomicInteger(0);

    final static AtomicInteger ranking = new AtomicInteger(1);
    final private int id;

    public Player(final int id) {
        this.id = id;
    }

    @Override
    public void run() {
        System.out.println("Player" + id + ": starting");
        final Random r = new Random();

        boolean isRunning = true;
        while (isRunning) {
            int current;
            int update;
            do {
                current = shared.get();
                if (current > 1000000) {
                    isRunning = false;
                    // reset
                    update = 0;
                } else {
                    update = current + 1 + r.nextInt(5);
                    isRunning = true;
                }
            } while (!shared.compareAndSet(current, update));
        }

        // Remark: just for visualization purposes.
        System.out.println("Player" + id + ": finished. Rank: " + ranking.getAndIncrement());
    }
}
```