Exercise 1

The starting point of all following versions is the implementation of the program's logic without any protection mechanism.

```java
class Sensor implements Runnable {
  final int threshold;

  public Sensor (final int threshold) {
    this.threshold = threshold;
  }

  @Override
  public void run() {
    System.out.println("Sensor[" + threshold + "]: start monitoring!");
    while (!A4Exercise1.resetIfAbove(threshold)) { /* Busy wait */ }
    System.out.println("Sensor [" + threshold + "]: threshold passed!");
  }
}

public class A4Exercise1 {
  private static int amount = 0;

  static int incrementAndGet(final int step) {
    amount += step;
    return amount;
  }

  static boolean resetIfAbove(final int threshold) {
    if (amount > threshold) {
      amount = 0;
      return true;
    }
    return false;
  }

  public static void main(final String[] args) {
    final List<Thread> threads = new ArrayList<>();

    for (int i = 1; i <= 10; i++) {
      final int sensorThreshold = (i * 10);
      threads.add(new Thread(new Sensor(sensorThreshold)));
    }

    for (final Thread t: threads)
      t.start();

    while (true) {
      final int increment = ThreadLocalRandom.current().nextInt(1, 9);
      final int newAmount = incrementAndGet(increment);
      System.out.println("Actuator: shared state incremented to " + newAmount);
      if (newAmount > 120)
          break;
      try {
        Thread.sleep(ThreadLocalRandom.current().nextLong(5, 11));
      } catch (final InterruptedException e) {
        e.printStackTrace();
      }
    }

    for (final Thread t : threads) {
      try {
        t.join();
      } catch (final InterruptedException e) {
      }
    }
  }
}
```

Volatile variabile alternative

```
private static volatile int amount = 0;
```

By specifying the *amount* variable as volatile, it is only possible to remove visibility problems. The Sensor threads terminate their execution as soon as their thresholds get reached, effectively reacting to the increment of the shared amount variable. Unfortunately, there are still race conditions left that the use of volatile variables cannot solve. The first is a ReadModifyWrite race condition in the *incrementAndGet* method when the value is incremented:

```
static int incrementAndGet(final int step) {
  amount += step;
  return amount;
}
```

While the second one is a CheckThenAct race condition located in the *resetIfAbove* method:

```
static boolean resetIfAbove(final int threshold) {
  if (amount > threshold) {
    amount = 0;
    return true;
  }
  return false;
}
```

Atomic variable without CAS

By modifying the *amount* variable to AtomicInteger and replacing the increment logic with the *addAndGet* method call, it is possible to fix the ReadModifyWrite race condition. The other race condition cannot be corrected without the implementation of the CAS logic.

```
private static AtomicInteger amount = new AtomicInteger();

static int incrementAndGet(final int step) {
  return amount.addAndGet(step);
}

static boolean resetIfAbove(final int threshold) {
  if (amount.get() > threshold) {
    amount.set(0);
    return true;
  }
  return false;
}
```

Atomic variable and CAS

By exploiting the compareAndSet method, it is finally possible to remove the CheckThenAct race condition too:

```
static boolean resetIfAbove(final int threshold) {
    int oldValue;
    do {
        oldValue = amount.get();
        // Threshold not reached: return false
        if (oldValue <= threshold)
            return false;

        // Threshold reached: proceed to reset.
        // In case value has changed in the meantime (cas fails) operation needs to be repeated
    } while (!amount.compareAndSet(oldValue, 0));
    return true;
}
```

To recap: all proposed versions solve the visibility-related issues, but only the atomic variables in combination with CAS provide a solution to all race conditions.

Exercise 2

The program has visibility problems present on the *completed*, *lat*, and *lon* variables and concurrent access on the *lat* and *lon* variables of the Coordinates class. Both states are written outside the lock-protected portion of the code, so there is no guarantee that the information will be visible from the main thread, when it calculates the distance between two coordinates. Moreover, the *lat* and *lon* values could modified while reading them, for example, because of a context switch during the execution of the distance method.

Solution using Immutable object

First, the *Coordinate* class needs to be transformed into an immutable object. All fields must be declared as private final and the class itself must be declared as final (to avoid that it gets extended).

```java
final class Coordinate {
  private final double lat;
  private final double lon;

  public Coordinate(final double lat, final double lon) {
    this.lat = lat;
    this.lon = lon;
  }

  /**
   * Returns the distance (expressed in km) between two coordinates
   */
  public double distance(final Coordinate from) {
    final double dLat = Math.toRadians(from.lat - this.lat);
    final double dLng = Math.toRadians(from.lon - this.lon);
    final double a = Math.sin(dLat / 2) * Math.sin(dLat / 2)
        + Math.cos(Math.toRadians(from.lat))
        * Math.cos(Math.toRadians(this.lat)) * Math.sin(dLng / 2)
        * Math.sin(dLng / 2);
    return (6371.000 * 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a)));
  }

  @Override
  public String toString() {
    return "[" + lat + ", " + lon + "]";
  }
}
```

Then, we need to correct the *run* method of the GPS class, by replacing the writes to the *lat* and *lon* variables with an invocation of the *Coordinate* constructor (by providing the two values as arguments). The construction of the new coordinate is executed outside the lock to avoid keeping the lock unnecessarily busy during the construction time of the object. The lock is therefore used exclusively for the publication of the new instance.

```java
class GPS implements Runnable {
  @Override
  public void run() {
    while (!S4Esercizio4.completed) {
      final double lat = ThreadLocalRandom.current().nextDouble(-90.0, +90.0);
      final double lon = ThreadLocalRandom.current().nextDouble(-180.0, +180.0);
      final Coordinate newCoordinate = new Coordinate(lat, lon);

      S4Esercizio4.lock.lock();
      try {
        S4Esercizio4.curLocation = newCoordinate;
      } finally {
        S4Esercizio4.lock.unlock();
      }
      // Wait before updating position
      try {
        Thread.sleep(ThreadLocalRandom.current().nextLong(1, 5));
      } catch (final InterruptedException e) {
        Thread.currentThread().interrupt();
      }
    }
    }
}
```

Finally, the visibility problem related to the *completed* flag must be fixed. This can be done by specifying it as volatile :

```
public class A4Exercise4 {
  static volatile boolean completed = false;
...
```

The solution could be further simplifier by declaring the *curLocation* reference as volatile (or AtomicReference) and removing the required lock to guarantee safe publication.

Exercise 3

```java
import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

final class ExchangeRates {
  private final double[][] rates = new double[5][5];

  // CHF, EUR, USD, GBP e JPY
  public static final int CHF = 0;
  public static final int EUR = 1;
  public static final int USD = 2;
  public static final int GBP = 3;
  public static final int JPY = 4;

  public ExchangeRates() {
    final Random random = new Random();
    final double[] tmp = new double[5];
    tmp[CHF] = random.nextDouble() * .5 + 1.;
    tmp[EUR] = random.nextDouble() * .5 + 1.;
    tmp[USD] = random.nextDouble() * .5 + 1.;
    tmp[GBP] = random.nextDouble() * .5 + 1.;
    tmp[JPY] = random.nextDouble() * .5 + 1.;

    for (int from = 0; from < 5; from++)
      for (int to = 0; to < 5; to++)
        rates[from][to] = tmp[from] / tmp[to];
  }

  final public double getExchangeRate(final int from, final int to) {
    // REMARK: immutable object -> cannot return reference!
    try {
      return rates[from][to];
    } catch (IndexOutOfBoundsException e) {
      return Double.NaN;
    }
  }

  final static String getCurrencyLabel(final int code) {
    switch (code) {
    case CHF:
      return "chf";
    case EUR:
      return "eur";
    case USD:
      return "usd";
    case GBP:
      return "gbp";
    case JPY:
      return "jpy";
    }
    return "";
  }
}

class Office implements Runnable {
  private final int id;

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

  @Override
  public void run() {
    final Random random = new Random();

    // Used to format output of exchange value and rate
    final DecimalFormat format_money = new DecimalFormat("000.00");
    final DecimalFormat format_rate = new DecimalFormat("0.00");

    while (A4Exercise3.isRunning) {
      final int from = random.nextInt(5);
      int to;
      do {
        to = random.nextInt(5);
      } while (to == from);

      final ExchangeRates newRates;
      A4Exercise3.lock.lock();
      try {
        newRates = A4Exercise3.currentRates;
      } finally {
        A4Exercise3.lock.unlock();
      }

      final double amount = random.nextInt(451) + 50;
      final double rate = newRates.getExchangeRate(from,to);
      final double changed = amount * rate;

      System.out.println("Office" + id + ": changed " + format_money.format(amount) + " "
          + ExchangeRates.getCurrencyLabel(from) + " in " + format_money.format(changed) + " "
          + ExchangeRates.getCurrencyLabel(to) + ". Rate: " + format_rate.format(rate));

      try {
        Thread.sleep(random.nextInt(4) + 1);
      } catch (final InterruptedException e) {
        // Ignored
      }
    }
  }
}

public class A4Exercise3 {
  static volatile boolean isRunning = true;
  final static Lock lock = new ReentrantLock();
  static ExchangeRates currentRates = new ExchangeRates();

  public static void main(final String[] args) {
    final List<Thread> allThread = new ArrayList<>();
    for (int i = 0; i < 10; i++) {
      allThread.add(new Thread(new Office(i)));
    }

    for (final Thread t : allThread)
      t.start();

    for (int i = 0; i < 100; i++) {
      final ExchangeRates newRates = new ExchangeRates();
      lock.lock();
      try {
        A4Exercise3.currentRates = newRates;
      } finally {
        lock.unlock();
      }
      System.out.println("New exchange rates available");
      try {
        Thread.sleep(100);
      } catch (final InterruptedException e) {
        e.printStackTrace();
      }
    }

    // Terminate simulation
    isRunning = false;

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

Exercise 4

When replacing implicit locks (synchronized keyword in the method declaration) with explicit locks (ReentrantLock) the execution time results very similar. However, by introducing read-write locks the execution time is shorter. Remark that with read-write-locks many more reads are performed compared to the other versions. The advantage of protecting a shared state using read-write locks is that the reads can occur simultaneously, as long as there are no writes in progress. In general, read-write locks perform well when the number of reads is significantly higher than the number of writes.

		Synchronized Method	**Reentrant Lock**	**Reentrant ReadWriteLock**
Thread execution time	(avg)	9994.1 ms	9682.8 ms	7380.0 ms
Number of reads	(avg)	899.7	829.3	3726.9
Number of writes	(avg)	73.8	78.9	90.0
Simulation time		10053 ms	9919 ms	8472 ms

```
// Reentrant lock
private static final ReentrantLock lock = new
ReentrantLock();

  public static long increment() {
    lock.lock();
    try {
      value++;
// Simulates the cost of a write operation
      Thread.sleep(10);
      return value;
    } catch (final InterruptedException e) {
      return value;
    } finally {
      lock.unlock();
    }
  }

  public static long getValue() {
    lock.lock();
    try {
// Simulates the cost of a read operation
      Thread.sleep(1);
      return value;
    } catch (final InterruptedException e) {
      return value;
    } finally {
      lock.unlock();
    }
  }
```

```
// Reentrant ReadWrite locks
  private static final ReentrantReadWriteLock
rwlock = new ReentrantReadWriteLock();

  public static long increment() {
    rwlock.writeLock().lock();
    try {
      value++;
// Simulates the cost of a write operation
      Thread.sleep(10);
      return value;
    } catch (final InterruptedException e) {
      return value;
    } finally {
      rwlock.writeLock().unlock();
    }
  }

  public static long getValue() {
    rwlock.readLock().lock();
    try {
// Simulates the cost of a read operation
      Thread.sleep(1);
      return value;
    } catch (final InterruptedException e) {
      return value;
    } finally {
      rwlock.readLock().unlock();
    }
  }
```