Soluzione esercizio 1

Sostituendo i lock impliciti (parola chiave synchronized nella dichiarazione del metodo) con lock espliciti (ReentrantLock) i tempi di esecuzione sono molto simili. Invece, introducendo i read-write locks il tempo è minore. Da notare che con i read-write locks vengono fatte molte più letture e più scritture. Il vantaggio di proteggere uno stato condiviso utilizzando i read-write locks è che le letture possono avvenire contemporaneamente, a patto che non vi siano scritture in corso. In generale, i read-write locks permettono di ottenere buone performances quando il numero di letture è molto maggiore rispetto al numero di scritture.

		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++;
// Questo serve a simulare il costo di una
scrittura
      Thread.sleep(10);
      return value;
    } catch (final InterruptedException e) {
      return value;
    } finally {
      lock.unlock();
    }
  }

  public static long getValue() {
    lock.lock();
    try {
// Questo serve a simulare il costo di una
lettura
      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++;
// Questo serve a simulare il costo di una
scrittura
      Thread.sleep(10);
      return value;
    } catch (final InterruptedException e) {
      return value;
    } finally {
      rwlock.writeLock().unlock();
    }
  }

  public static long getValue() {
    rwlock.readLock().lock();
    try {
// Questo serve a simulare il costo di una
lettura
      Thread.sleep(1);
      return value;
    } catch (final InterruptedException e) {
      return value;
    } finally {
      rwlock.readLock().unlock();
    }
  }
```

Soluzione esercizio 2

L'esercizio richiede di trasformare la classe Sensore per sfruttare l'operazione di compare-and-set delle variabili atomiche. Il motivo è dovuto al fatto che la logica del sensore contiene una compound action racchiusa nel metodo *resetIfAbove* che nell'implementazione fornita non è ancora resa atomica. Tra la lettura del valore attuale, il confronto con la soglia e l'eventuale reset, il thread può essere interrotto, causando delle race condition di tipo check-then-act. Potrebbe cioè accadere che più thread riescano a vedere superata la propria soglia ed eseguire il reset.

Per rendere atomica l'operazione, si sostituisce l'operazione di set con l'operazione di compare-and-set: l'operazione di compare-and-set riesce solo se il valore dello stato condiviso (*S4Esercizio2.counter*) corrisponde ancora allo stato iniziale (*currentAmount*). Se il valore nel frattempo è cambiato, perché il main thread è riuscito ad aumentare il valore oppure un altro sensore è riuscito a resettare il contatore, bisogna ripetere l'operazione partendo dal nuovo valore dello stato condiviso. Per quest'ultimo motivo tutta l'operazione viene racchiusa in un ciclo do-while. Con questa modifica garantiamo che solo un sensore alla volta riesca a completare l'operazione di reset.

```
private boolean resetIfAbove() {
  int currentAmount;
  do {
    currentAmount = S4Esercizio2.counter.get();
    if (currentAmount < soglia)
      return false;
  } while (!S4Esercizio2.counter.compareAndSet(currentAmount, 0));
  return true;
}
```

Soluzione esercizio 3

Il programma presenta un problema di visibilità della memoria: tutte le modifiche eseguite dal main thread sulle variabili condivise (*home*, *office*, *mobile*, *emergency*, *version*) non vengono viste dai *Contact* threads.

La soluzione proposta evita del tutto l'uso di lock impliciti ed espliciti, sfruttando la proprietà dell'estensione della visibilità della memoria per le variabili volatile. Tutte le variabili visibili da un thread prima della scrittura di una variabile volatile diventano visibili ad un altro thread che esegue una lettura della stessa variabile volatile.

In questo caso, visto che il main thread scrive sempre nello stesso ordine le variabili *home*, *office*, *mobile* ed *emergency* prima della variabile *version*, e tutti i *Contact* threads leggono come prima variabile condivisa proprio la variabile *version*, si può sfruttare l'effetto di estensione della visibilità della memoria delle variabili volatile, dichiarando unicamente la variabile *version* come volatile. Appena il main thread cambia il valore di *version*, tutti i thread vedranno sia il nuovo valore della variabile *version* sia il valore aggiornato di *home*, *office*, *mobile* ed *emergency*.

```
public class S4Esercizio3 {

        // 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;


...
}
```

Soluzione esercizio 4

Il programma ha problemi di visibilità per le variabili *completed, lat* e *lon* e di accessi concorrenti per le variabili *lat* e *lon* della classe *Coordinate*. Le due informazioni vengono infatti scritte all'esterno della porzione di codice protetta da lock e quindi non c'è alcuna garanzia che le informazioni siano visibili dal main thread quando calcola la distanza tra due coordinate. Inoltre i valori di *lat* e *lon* potrebbero cambiare mentre li sta leggendo, ad esempio a causa di un context switch durante l'esecuzione del metodo *distance*.

Soluzione con oggetto immutable

Per prima cosa è necessario trasformare la classe *Coordinate* in un oggetto immutable. Tutti i campi devono essere dichiarati come private final e la classe stessa deve essere dichiarata come final (per evitare che possa essere estesa).

```
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 + "]";
  }
}
```

Successivamente bisogna correggere il metodo *run* della classe GPS, sostituendo le scritture alle variabili *lat* e *lon* con l'invocazione al costruttore *Coordinate* (passandogli i due valori). La costruzione della nuova coordinata viene fatta fuori dal lock, in modo da usare il lock esclusivamente per la pubblicazione della nuova istanza, evitando così di tenere inutilmente occupato il lock per il tempo di costruzione dell'oggetto.

```
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();
```

```
        }
      }
    }
}
```

Infine bisogna correggere il problema di visibilità legato alla flag di completamento *completed* specificandola ad esempio come volatile.

```
public class S4Esercizio4 {
  static volatile boolean completed = false;
...
```

La soluzione potrebbe inoltre essere semplificata dichiarando la referenza *curLocation* come volatile (o AtomicReference) e togliendo di conseguenza la necessità del lock per la pubblicazione degli oggetti.

Soluzione esercizio 5

```
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 TassiDiCambio {
  private final double[][] tassi = 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 TassiDiCambio() {
    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++)
        tassi[from][to] = tmp[from] / tmp[to];
  }

  final public double getExchangeRate(final int from, final int to) {
    // REMARK: immutable object -> cannot return reference!
    try {
      return tassi[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 Sportello implements Runnable {
  private final int id;

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

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

    // Usato per formattare l'output della valuta e tasso di cambio
    final DecimalFormat format_money = new DecimalFormat("000.00");
    final DecimalFormat format_tasso = new DecimalFormat("0.00");

    while (S4Esercizio5.isRunning) {
      final int from = random.nextInt(5);
```

```
      int to;
      do {
        to = random.nextInt(5);
      } while (to == from);

      final TassiDiCambio nuoviTassi;
      S4Esercizio5.lock.lock();
      try {
        nuoviTassi = S4Esercizio5.tassiAttuali;
      } finally {
        S4Esercizio5.lock.unlock();
      }

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

      System.out.println("Sportello" + id + ": ho cambiato " + format_money.format(amount) + " "
          + TassiDiCambio.getCurrencyLabel(from) + " in " + format_money.format(changed) + " "
          + TassiDiCambio.getCurrencyLabel(to) + " tasso " + format_tasso.format(tasso));

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

public class S4Esercizio5 {
  static volatile boolean isRunning = true;
  final static Lock lock = new ReentrantLock();
  static TassiDiCambio tassiAttuali = new TassiDiCambio();

  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 Sportello(i)));
    }

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

    for (int i = 0; i < 100; i++) {
      final TassiDiCambio nuoviTassi = new TassiDiCambio();
      lock.lock();
      try {
        S4Esercizio5.tassiAttuali = nuoviTassi;
      } finally {
        lock.unlock();
      }
      System.out.println("Nuovi tassi di cambio disponibili");
      try {
        Thread.sleep(100);
      } catch (final InterruptedException e) {
        e.printStackTrace();
      }
    }

    // Termina simulazione
    isRunning = false;

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