Soluzione esercizio 1

Analizzando il programma solo il main thread (che rappresenta la sede centrale di cambio) si occupa di creare un nuovo oggetto per i nuovi tassi di cambio, mentre i threads che rappresentano gli sportelli vi accedono in sola lettura. L'effetto di corretta visibilità della memoria, ottenuto grazie alla referenza volatile utilizzata per pubblicare l'oggetto, combinato con l'accesso in sola lettura (dopo la pubblicazione), rendono l'oggetto effectively immutable. Non è indispensabile che l'oggetto venga reso completamente immutable o che si introducano ulteriori meccanismi di sincronizzazione.

```
import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

class TassiDiCambio {
  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];
  }

  public 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 (S5Esercizio3.isRunning) {
      final int from = random.nextInt(5);
      int to = random.nextInt(5);
      if (to == from)
        to = (to + 3) % 5;

      final double amount = random.nextInt(451) + 50;
```

```
      final double tasso = S6Esercizio1.tassiAttuali.tassi[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) {
      }
    }
  }
}

public class S6Esercizio1 {
  static volatile boolean isRunning = true;
  static volatile 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++) {
      S6Esercizio1.tassiAttuali = new TassiDiCambio();
      System.out.println("Nuovi tassi di cambio disponibili");
      try {
        Thread.sleep(100);
      } catch (final InterruptedException e) {
      }
    }

    // Termina simulazione
    isRunning = false;

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

Soluzione esercizio 2

Il programma presenta un problema legato alla porzione di codice protetta da lock. Ogni persona che vuole andare al bagno ottiene l'accesso esclusivo a tutti i bagni, impedendo agli altri di verificare se ci sono bagni liberi. Quindi, il lock protegge una porzione di codice troppo grande e va spezzato in più lock indipendenti tra di loro. Per trovare un bagno libero è necessario conoscere lo stato di un singolo bagno e non di tutti quelli disponibili!

Soluzione con lock-splitting

La soluzione proposta rimpiazza il lock unico per tutti i bagni con un lock indipendente per ogni bagno, spostando la keyword synchronized dal metodo *occupaBagno* ai metodi *provaOccupare* e *libera* della classe Bagno.

```
class Bagno {
  private boolean occupato = false;

  // Ritorna false se gia occupato
  public synchronized boolean provaOccupare() {
    if (occupato)
      return false;
    this.occupato = true;
    return true;
  }

  public synchronized void libera() {
    this.occupato = false;
  }
}
```

Soluzione con AtomicBoolean

Si può risolvere il problema, rinunciando completamente ai lock, dichiarando la variabile occupato come AtomicBoolean. A questo punto è possibile rimuovere la keyword synchronized dal metodo *occupaBagno*. In seguito, va rifattorizata la classe Bagno. Il metodo *provaOccupare* sfrutta l'operazione di CAS per provare ad occupare il bagno, mentre il metodo libera reimposta la variabile a false.

```
class BagnoAtomic {
  private final AtomicBoolean occupato = new AtomicBoolean(false);

  // Ritorna false se gia occupato
  public boolean provaOccupare() {
    return occupato.compareAndSet(false, true);
  }

  public void libera() {
    occupato.set(false);
  }
}
```