Solution exercise 1

The analysis of the program shows that only the main thread (representing the exchange headquarters) is responsible for creating a new object for the exchange rates. The other threads (representing the counters) access the shared exchange rates in read-only mode. The effect of correct memory visibility, achieved thanks to the volatile reference used to publish the object, combined with read-only access (after publication), makes the object effectively immutable. Therefore, it is not required that the object is implemented as fully immutable or that additional synchronization mechanisms are introduced.

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

class ExchangeRates {
    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 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 the output of the currency and exchange rate
        final DecimalFormat format_money = new DecimalFormat("000.00");
        final DecimalFormat format_tasso = new DecimalFormat("0.00");

        while (A7Exercise1.isRunning) {
            final int from = random.nextInt(5);
```

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

            final double amount = random.nextInt(451) + 50;
            final double rate = A7Exercise1.currentExchangeRates.rates[from][to];
            final double changed = amount * rate;

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

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

public class A7Exercise1 {
    static volatile boolean isRunning = true;
    static volatile ExchangeRates currentExchangeRates = 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();
            A7Exercise1.currentExchangeRates = newRates;

            System.out.println("---------------------------");
            System.out.println("New exchange rate available!");
            System.out.println("---------------------------");
            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();
            }
    }
}
```

Solution exercise 2

The program has a problem related to the lock-protected region of the code. Each person that wants to go to the bathroom gets exclusive access to all bathrooms, preventing others from checking if there is a bathroom available. Therefore, the lock protects a portion of code, which is too large and has to be split into several independent locks. To find a free bathroom it is needed to know the status of a single bathroom and not of all the available ones!

Lock-splitting solution

The proposed solution replaces the unique lock for all bathrooms with an independent lock for each bathroom, moving the synchronized keyword from the occupy method to the *tryToOccupy* and *leave* methods of the Restroom class.

```
class Restroom {
    private boolean isOccupied = false;

    // Return false in case restroom is already occupied
    public synchronized boolean tryToOccupy() {
        if (isOccupied)
            return false;
        this.isOccupied = true;
        return true;
    }

    public synchronized void leave() {
        this.isOccupied = false;
    }
}
```

AtomicBoolean solution

The problem can be solved without using any locks, by declaring the *isOccupied* variable as AtomicBoolean. At this point, the keyword synchronized can be removed from the *tryToOccupy* and *leave* methods. Then, the Restroom class must be refactored. The tryToOccupy method takes advantage of the CAS operation to try to occupy the bathroom, while the leave method resets the variable to false.

```
class Restroom {
    final private AtomicBoolean isOccupied = new AtomicBoolean(false);

    // Return false if the restroom is already occupied
    public boolean tryToOccupy() {
        return this.isOccupied.compareAndSet(false, true);
    }

    public void leave() {
        this.isOccupied.set(false);
    }
}
```

Solution exercise 3

The program is composed of an object of type Rectangle shared with a static rect reference and 5 Resizer threads which are responsible for modifying its size. The Rectangle class is composed of an immutable state (the two variables x1 and y1 are final) and a mutable state (x2 and y2). For the mutable state, the program has both visibility and race conditions problems. Between the invocations of the getX2() and getY2() methods, the shared state might be modified by another thread, with consequent potentially inconsistent execution. Similar problems might happen when the setX2() and setY2() methods are called to modify the state. These operations have to be managed as compound actions: to maintain a consistent state it is required to read or write both values (x2 and y2) as a single atomic operation. Instead, the operations performed to validate the transformation (isPoint, isLine, isNegative) are not subject to concurrency problems because access is made to local and final variables, therefore without risk of visibility problems or concurrent modifications on the shared state.

To solve the problems there are some alternative solutions. The most immediate one is to introduce a lock (implicit or explicit) to protect the operations of reading and writing the shared mutable state. However, this solution is not efficient because it introduces the overhead of the locks and also makes all reading operations synchronous (they cannot be performed in parallel). Replacing the lock (implicit or explicit) with a ReadWriteLock improves the performances because it allows more read operations in parallel. The solution that provides better performance can be obtained by converting the Rectangle class into an immutable class.

Immutable Object

To make the Rectangle class immutable, it is required to declare the class and the x2 and y2 instance variables final. In addition, the setX2() and setY2() methods have to be replaced with a single resize() method that takes care of the entire resize operation. This method has to return a new instance of the Rectangle class if the operation is valid, or null if the transformation is not possible. As a further refinement, the static reference has to be replaced with an AtomicReference and the reference substitution has to be performed using the CAS idiom. This solution achieves great performance because it reduces the synchronization overhead between threads.

```
final class Rectangle {
  final int x1;
  final int y1;
  final int x2;
  final int y2;

  public Rectangle(final int newX1, final int newY1, final int newX2, final int newY2) {
    x1 = newX1;
    y1 = newY1;
    x2 = newX2;
    y2 = newY2;
  }

  public int getX1() {
    return x1;
  }

  public int getX2() {
    return x2;
  }

  public int getY1() {
    return y1;
  }
```

```
  public int getY2() {
    return y2;
  }

  public Rectangle resize(final int deltaX2, final int deltaY2) {
    // compute new coordinates  x2 e y2
    final int newX2 = x2 + deltaX2;
    final int newY2 = y2 + deltaY2;

    final boolean isLine = (x1 == newX2) || (y1 == newY2);
    final boolean isPoint = (x1 == newX2) && (y1 == newY2);
    final boolean isNegative = (x1 > newX2) || (y1 > newY2);
    if (isLine || isPoint || isNegative)
        return null;
    return new Rectangle(x1, y1, newX2, newY2);
  }

  @Override
  public String toString() {
    return "[" + x1 + ", " + y1 + ", " + x2 + ", " + y2 + "]";
  }
}

class Resizer implements Runnable {
  @Override
  public void run() {
    final Random random = new Random();
    for (int i = 0; i < 1000; i++) {
      try {
        Thread.sleep(random.nextInt(2) + 2);
      } catch (final InterruptedException e) {
      }
      Rectangle startRect = null;
      Rectangle sizedRect = null;

      do {
        startRect = A7Exercise3.rect.get();
        // generate variation between -2 e 2
        final int deltaX2 = random.nextInt(5) - 2;
        final int deltaY2 = random.nextInt(5) - 2;

        sizedRect = startRect.resize(deltaX2, deltaY2);
      } while (sizedRect == null || !A7Exercise3.rect.compareAndSet(startRect, sizedRect));
    }
  }
}

/**
 * Program that continuously simulates the variation of the rectangle's dimensions
 */
public class A7Exercise3 {
  final static AtomicReference<Rectangle> rect = new AtomicReference<Rectangle>(new Rectangle(10,
10, 20, 20));

  public static void main(final String[] args) {
    final List<Thread> allThreads = new ArrayList<Thread>();
    for (int i = 0; i < 5; i++)
      allThreads.add(new Thread(new Resizer()));
    System.out.println("Simulation started");
    for (final Thread t : allThreads) {
      t.start();
    }

    for (final Thread t : allThreads) {
      try {
        t.join();
      } catch (final InterruptedException e) {
        e.printStackTrace();
      }
    }
    System.out.println("Simulation finished");
  }
}
```

Solution exercise 4

```
enum Color {
    YELLOW, BLUE, GREEN, PINK;

    public static Color getRandom() {
        final Color[] vals = Color.values();
        final int index = ThreadLocalRandom.current().nextInt(vals.length);
        return vals[index];
    }
}

class PostIt {
    final private Color color;
    private long expireTime;
    private String text;

    public PostIt(final Color color) {
        this.color = color;
    }

    public void attach(final String text, final long duration) {
        this.text = text;
        this.expireTime = System.currentTimeMillis() + duration;
    }

    public boolean isExpired() {
        return System.currentTimeMillis() > expireTime;
    }

    @Override
    public String toString() {
  return String.format("%s PostIt msg='%s' expired=%s", color, text, Boolean.toString(isExpired()));
    }
}

class User implements Runnable {
    final int id;
    public User(final int id) {
        this.id = id;
    }

    public void run() {
        long cnt = 0;
        long nullCounter = 0;
        while (S7Esercizio4.isRunning) {
            final PostIt current = pickRandomPostIt();

            if (current == null) {
                nullCounter++;
            }else{
                current.attach(String.format("User%d message %d", id, cnt++),
ThreadLocalRandom.current().nextLong(30, 50));
                PostItExample.WHITEBOARD.add(current);
                if ((cnt % 100) == 0)
                    System.out.println(String.format("User%d attached %d postIts (nullCounter=%d)",
id, cnt, nullCounter));
            }

            try {
                TimeUnit.MILLISECONDS.sleep(ThreadLocalRandom.current().nextLong(12, 58));
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        System.out.println(String.format("User%d terminating. Attached %d postIts (nullCounter=%d)",
id, cnt, nullCounter));
    }

    private PostIt pickRandomPostIt() {
        // Choose Random
        final Color choosen = Color.getRandom();
        // Check if available, otherwise skip
        final Queue<PostIt> postItBlock = S7Esercizio4.DISPENSER.get(choosen);
        if (postItBlock == null)
            return null;
```

```
        final PostIt current = postItBlock.poll();
        if (postItBlock.isEmpty()) {
            // REMARK: use remove(key, value) method to avoid check then act!!
            // postItBlock is empty, but secretary may have already replaced it with a new one!
            // Using the remove(key) method may remove the new block instead of the empty one!
            S7Esercizio4.DISPENSER.remove(choosen, postItBlock);
        }
        return current;
    }
}

public class S7Esercizio4 {

    private static Queue<PostIt> createPostItBlock(final Color color) {
        final Queue<PostIt> block = new ConcurrentLinkedDeque<>();
        int numPostIt = ThreadLocalRandom.current().nextInt(55, 156);

        for (int i = 0; i < numPostIt; i++)
            block.add(new PostIt(color));

        return block;
    }

   public static volatile boolean isRunning = true;

    public static final ConcurrentHashMap<Color, Queue<PostIt>> DISPENSER = new
ConcurrentHashMap<>();

    public static final CopyOnWriteArrayList<PostIt> WHITEBOARD = new CopyOnWriteArrayList<>();

    public static void main(String[] args) {

        final Map<Color, Queue<Queue<PostIt>>> stock = new HashMap<>();

        // Init stock
        for (final Color color : Color.values()) {
            final int numColorStock = ThreadLocalRandom.current().nextInt(1, 7);
            final Queue<Queue<PostIt>> colorStock = new ArrayDeque<>();
            for (int count = 0; count < numColorStock; count++) {
                colorStock.add(createPostItBlock(color));
            }
            stock.put(color, colorStock);
        }

        // Init dispenser
        for (final Color color : Color.values()) {
            DISPENSER.put(color, createPostItBlock(color));
        }

        final List<Thread> allThreads = new ArrayList<>();
        for (int i = 1; i <= 5; i++) {
            allThreads.add(new Thread(new User(i)));
        }

        allThreads.forEach(Thread::start);

        int print = 0;
        while (isRunning) {

            try {
                TimeUnit.MILLISECONDS.sleep(ThreadLocalRandom.current().nextLong(9,  24));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            int replacements = checkDispenser(stock);
            if (replacements > 0) {
                System.out.println("Replaced " + replacements + " PostIT blocks");
            }

            // Clear whiteboard
            if (WHITEBOARD.removeIf(PostIt::isExpired)) {
                System.out.println("Whiteboard has been cleared from expired notes");
            }

            if (++print > 40) {
```

```
                System.out.println(WHITEBOARD.parallelStream()
                                             .map(Object::toString)
                                             .collect(Collectors.joining(", ")));
                print = 0;
            }

            isRunning = (stock.isEmpty() == false);
        }
    }

    private static int checkDispenser(final Map<Color, Queue<Queue<PostIt>>> stock) {
        int cnt = 0;
        for (final Color color : Color.values()) {
            if (!stock.containsKey(color))
                continue;
            final Queue<Queue<PostIt>> blockColorStock = stock.get(color);
            if (blockColorStock.isEmpty()) {
                stock.remove(color);
                continue;
            }
            final Queue<PostIt> replacement = blockColorStock.poll();
            Queue<PostIt> previousValue = DISPENSER.putIfAbsent(color, replacement);
            if (previousValue == null)
                cnt++;
            else
                // Did not replace: put back to stock
                blockColorStock.add(replacement);
        }
        return cnt;
    }
}
```