Domanda 1 - Sostituzione locks con variabili atomiche

Per sfruttare le variabili atomiche al posto dei lock espliciti va dichiarata la variabile *popolazione* come *AtomicLongArray*. Inoltre, va riscritta la logica dell'incremento e della decimazione della popolazione utilizzando l'idioma del CAS all'interno di un ciclo do-while.

```
/**
 * Classe che simula periodi di carestia per i paesi
 */
class Carestia implements Runnable {
  @Override
  public void run() {
    final Random random = new Random();
    // ad ogni giro introduce carestia per un determinato paese scelto a caso
    for (int i = 0; i < 50; i++) {
      final int paeseScelto = random.nextInt(EsercizioPopolazioni.popolazione.length());
      final double fattoreDecimazione = random.nextDouble() * 0.6 + 0.1;

      // decima la popolazione
      long numeroAbitanti;
      long numeroAbitantiIniziale;
      do {
        numeroAbitantiIniziale = EsercizioPopolazioni.popolazione.get(paeseScelto);
        numeroAbitanti = (long) (numeroAbitantiIniziale * fattoreDecimazione);
      } while (!EsercizioPopolazioni.popolazione.compareAndSet(paeseScelto, numeroAbitantiIniziale,
numeroAbitanti));

      System.out.println("Carestia: Popolazione " + paese Scelto + " diminuita a "
          + numeroAbitanti + " del fattore " + fattoreDecimazione);

      try {
        // pausa fra un periodo di carestia e l'altro
        Thread.sleep(100);
      } catch (final InterruptedException e) {
      }
    }
  }
}

/**
 * Classe che simula periodi di prosperita per i paesi
 */
class Prosperita implements Runnable {
  @Override
  public void run() {
    final Random random = new Random();
    // ad ogni giro introduce prosperità per un determinato paese scelto a caso
    for (int i = 0; i < 100; i++) {
      final int paeseScelto = random.nextInt(EsercizioPopolazioni.popolazione.length());
      final double fattoreCrescita = random.nextDouble() * 0.55 + 1.05;

      // incrementa la popolazione
      long numeroAbitanti;
      long numeroAbitantiIniziale;
      do {
        numeroAbitantiIniziale = EsercizioPopolazioni.popolazione.get(paeseScelto);
        numeroAbitanti = (long) (numeroAbitantiIniziale * fattoreCrescita);
      } while (!EsercizioPopolazioni.popolazione.compareAndSet(paeseScelto, numeroAbitantiIniziale,
numeroAbitanti));

      System.out.println("Prosperita: Popolazione " + paeseScelto
          + " cresciuta a " + numeroAbitanti + " del fattore "
          + fattoreCrescita);

      try {
        // pausa fra un periodo di prosperità e l'altro
        Thread.sleep(50);
      } catch (final InterruptedException e) {
      }
    }
  }
}
```

```
/**
 * Programma che simula la variazione demografica di 5 paesi
 */
public class EsercizioPopolazioni {
  static final AtomicLongArray popolazione = new AtomicLongArray(5);

  public static void main(final String[] args) {
    // la popolazione iniziale è di 1000 abitanti per ogni paese
    for (int i = 0; i < 5; i++)
      popolazione.set(i, 1000);

    final List<Thread> allThreads = new ArrayList<>();
    allThreads.add(new Thread(new Prosperita()));
    allThreads.add(new Thread(new Prosperita()));
    allThreads.add(new Thread(new Carestia()));

    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");
  }
}
```

Domanda 2

Il programma è composto da un oggetto di tipo *Rettangolo* condiviso tramite una referenza static *rect* e da 5 threads *Resizer* che si occupano di modificarne le dimensioni. La classe *Rettangolo* è composta da stato immutable (le 2 variabili x1 e y1 sono final) e da stato mutable (x2 e y2). Per lo stato mutable, il programma presenta sia problemi di visibilità che problemi di race conditions. Infatti, fra le invocazioni dei metodi *getX2*() e *getY2*(), l'istanza dell'oggetto condiviso potrebbe venire modificata da un altro thread rendendo l'esecuzione inconsistente. Lo stesso potrebbe accadere quando vengono chiamati i metodo *setX2()* e *setY2()* per modificarne lo stato. Queste operazioni sono quindi da considerarsi come compound actions: per mantenere uno stato consistente è necessario leggere o scrivere entrambi i valori (x2 e y2) in una sola operazione atomica. Infine le operazioni eseguite per validare la trasformazione (isPoint, isLine, isNegative) non presentano problemi in quanto accedono a variabili final che quindi non soffrono di problemi di visibilità o di modifiche concorrenti.

Per risolvere i problemi legati alle race conditions ci sono varie possibilità. La soluzione più immediata è quella di introdurre un lock (implicito o esplicito) per proteggere le operazioni di lettura e di scrittura di entrambe le variabili. Questo genere di soluzione non è però quella più performante in quanto introduce l'overhead dovuto ai locks ed inoltre rende sincrone tutte le operazioni di lettura (non possono avvenire in parallelo). Sostituendo il lock (implicito o esplicito) con un ReadWriteLock si migliorano le performances solo parzialmente poiché permetterebbe di effettuare più letture in parallelo. La soluzione che offre prestazioni migliori è quella di rendere immutable la classe Rettangolo.

Oggetto immutable

Per rendere immutable la classe Rettangolo bisogna dichiarare final la classe e le variabili d'istanza x2 ed y2. Inoltre, bisogna sostituire i metodi *setX2()* e *setY2()* con un unico metodo *resize()* che si occupi dell'intera operazione di ridimensionamento. Tale metodo deve restituire una nuova istanza della classe Rettangolo, qualora l'operazione fosse valida, o null se la trasformazione non fosse possibile. Come ulteriore

perfezionamento, per evitare che vengano perse delle trasformazioni valide, è necessario sostituire la referenza static con un'AtomicReference ed eseguire l'operazione di sostituzione della referenza sfruttando l'idioma del CAS. Questa soluzione permette di ottenere ottime performance perché riduce l'overhead di sincronizzazione tra i vari 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) {
    // calcola nuove coordinate 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 < 500; i++) {
      try {
        Thread.sleep(random.nextInt(2) + 2);
      } catch (final InterruptedException e) {
      }

      Rectangle startRect = null;
      Rectangle sizedRect = null;

      do {
        startRect = EsercizioRettangolo.rect.get();
        // genera variazione per punti tra -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 || !EsercizioRettangolo.rect.compareAndSet(startRect, sizedRect));

      System.out.println(sizedRect);
    }
  }
}

/**
 * Programma che simula variazioni continue delle dimensioni di un rettangolo
 */
public class EsercizioRettangolo {
  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");
  }
}
```

Domanda 3 – Grande Magazzino

```
class Prodotto {
  private static final AtomicInteger prodottiDisponibili = new AtomicInteger();
  private final int id;
  private final int prezzo;
  private int quantita;

  public Prodotto() {
    final Random r = new Random();
    this.prezzo = r.nextInt(5) + 1;
    this.quantita = r.nextInt(10) + 1;
    this.id = prodottiDisponibili.incrementAndGet();
  }

  public boolean acquista() {
    boolean stockFinito = false;

    synchronized (this) {
      // Prodotto finito
      if (quantita == 0)
        return false;
      // decrementa quantita
      quantita--;
      // verifica se era l'ultimo prodotto
      stockFinito = (quantita == 0);
    }
    // Non ho necessita di lock!
    if (stockFinito)
      prodottiDisponibili.decrementAndGet();
    return true;
  }

  public int getPrezzo() {
    return prezzo;
  }

  public static boolean prodottiDisponibili() {
    return prodottiDisponibili.get() > 0;
  }
```

```
  @Override
  public String toString() {
    final int cnt;
    synchronized (this) {
      cnt = quantita;
    }
    return "Prodotto" + id + " prezzo: " + prezzo + "$ quantita: " + cnt;
  }
}

class Cliente implements Runnable {
  private static final AtomicInteger clientiAttivi = new AtomicInteger();

  @Override
  public void run() {
    final int id = clientiAttivi.incrementAndGet();
    while (!Magazzini.isAperto()) {
      // Attendi apertura
    }

    final Random random = new Random();
    int soldi = 20;
    int tentativi = 0;

    while (true) {
      final Prodotto prodottoScelto = Magazzini.getProdotto(random.nextInt(Magazzini.NUM_PRODOTTI));
      final boolean disponibilitaEconomica = (soldi >= prodottoScelto.getPrezzo());
      if (disponibilitaEconomica && prodottoScelto.acquista()) {
        soldi -= prodottoScelto.getPrezzo();
        tentativi = 0;
      } else
        tentativi++;

      if (soldi == 0) {
        System.out.println("Cliente" + id + ": soldi finiti!");
        break;
      } else if (tentativi >= 10) {
        System.out.println("Cliente" + id + ": tentativi finiti!");
        break;
      } else if (!Magazzini.isAperto()) {
        System.out.println("Cliente" + id + ": negozio chiuso!");
        break;
      }
      // Attesa prima del prossimo acquisto
      try {
        Thread.sleep(1 + random.nextInt(5));
      } catch (final InterruptedException e) {
        e.printStackTrace();
      }
    }
    clientiAttivi.decrementAndGet();
    System.out.println("Cliente" + id + " termina. " + soldi + "$ tentativi: " + tentativi);
  }

  public static boolean checkClientiAttivi(final int numClienti) {
    return clientiAttivi.get() == numClienti;
  }
}

public class Magazzini {
  public static final int NUM_PRODOTTI = 10;
  private static final int NUM_CLIENTI = 10;
  private static final Prodotto[] prodotti = new Prodotto[NUM_PRODOTTI];
  private static volatile boolean aperto = false;

  public static boolean isAperto() {
    return aperto;
  }

  public static Prodotto getProdotto(final int index) {
    return prodotti[index];
  }

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

```
    for (int i = 0; i < NUM_PRODOTTI; i++)
      prodotti[i] = new Prodotto();
    for (int i = 0; i < NUM_CLIENTI; i++)
      allThreads.add(new Thread(new Cliente()));

    // Fai partire tutti i threads
    allThreads.forEach(Thread::start);

    // Attendi che tutti i clienti siano pronti
    while (!Cliente.checkClientiAttivi(NUM_CLIENTI))
      ;

    // Tutti i clienti sono pronti: apri negozio
    aperto = true;

    // Finche ci sono clienti, verifica ed eventualmente chiudi il negozio se sono finiti i prodotti
    while (!Cliente.checkClientiAttivi(0) && aperto) {
      aperto = Prodotto.prodottiDisponibili();
    }

    // Attendi terminazione dei threads
    for (final Thread t : allThreads)
      try {
        t.join();
      } catch (final InterruptedException e) {
        e.printStackTrace();
      }

    // Stampa prodotti
    System.out.println("================================================");
    Arrays.asList(prodotti).forEach(System.out::println);
  }
}
```