Soluzione esercizio 1

Soluzione con blocchi synchronized

Per ridurre al minimo il tempo di lock, ma continuare a garantire la thread safety, si può aggiungere un blocco synchronized all'interno di ogni blocco if. Di conseguenza, l'operazione di lock viene eseguita solo se viene letta o modificata la HashMap.

```
while (--cnt > 0) {
  final String key = getClass().getSimpleName() + random.nextInt(S7Esercizio1.NUM_WORKERS);
  updateCounter(random.nextBoolean());
   if (counter == 0) {
    synchronized (sharedMap) {
      if (sharedMap.containsKey(key) && sharedMap.get(key).equals(int1)) {
        sharedMap.remove(key);
        log("{" + key + "} remove 1");
      }
    }
  } else if (counter == 1) {
    synchronized (sharedMap) {
      if (!sharedMap.containsKey(key)) {
        sharedMap.put(key, int1);
        log("{" + key + "} put 1");
      }
    }
  } else if (counter == 5) {
    synchronized (sharedMap) {
      if (sharedMap.containsKey(key) && sharedMap.get(key).equals(10)) {
        final Integer prev = sharedMap.put(key, int5);
        log("{" + key + "} replace " + prev.intValue() + " with 5");
      }
    }
  } else if (counter == 10) {
    synchronized (sharedMap) {
      if (sharedMap.containsKey(key)) {
        final Integer prev = sharedMap.put(key, int10);
        log("{" + key + "} replace " + prev.intValue() + " with 10");
      }
    }
  }
}
```

Soluzione con Concurrent Collections

Utilizzando una ConcurrentHashMap, è possibile approfittare delle compound actions messe a disposizione dall'interfaccia ConcurrentMap e rimuovere la sincronizzazione introdotta con i blocchi synchronized. Per ottenere la thread-safety va sfruttata l'atomicità messa a disposizione dai metodi putIfAbset(), remove() e replace().

```
private final static ConcurrentHashMap<String, Integer> sharedMap
[...]                                                    = new ConcurrentHashMap<String, Integer>();
while (--cnt > 0) {
  final String key = getClass().getSimpleName() + random.nextInt(S7Esercizio1.NUM_WORKERS);
  updateCounter(random.nextBoolean());

  if (counter == 0) {
    if (sharedMap.remove(key, int1))
      log("{" + key + "} remove 1");
  } else if (counter == 1) {
    final Integer val = sharedMap.putIfAbsent(key, int1);
    if (val == null)
      log("{" + key + "} put 1");
  } else if (counter == 5) {
    if (sharedMap.replace(key, int10, int5))
      log("{" + key + "} replace " + int10 + " with 5");
  } else if (counter == 10) {
    final Integer prev = sharedMap.replace(key, int10);
    if (prev != null)
      log("{" + key + "} replace " + prev.intValue() + " with 10");
  }}
```

Soluzione esercizio 2

Il programma è composto da 15 threads in lettura (ReadWorker) e *dal main thread* che accedono ad una ArrayList condivisa (sharedPhrase). *Il main thread* aggiunge, ad intervalli regolari, una delle *6* stringhe contenute nell'array *nouns* all' ArrayList. I threads in lettura invece formano una stringa con le parole contenute nella sharedPhrase e la confrontano con una propria copia locale. Quando le due stringhe differiscono, viene sostituita la copia locale con la nuova stringa. Inoltre, ogni thread in lettura tiene traccia di quante letture riesce ad eseguire dall'ArrayList condivisa e quante frasi nuove riesce ad ottenere.

Se si esegue il programma, si possono notare diversi problemi: alcuni ReadWorker segnalano 0 come conteggio delle stringhe, mentre altri riescono a contare correttamente. Inoltre, a volte vengono lanciate delle ConcurrentModificationException. Per quanto riguarda il primo malfunzionamento, la causa è un problema di visibilità della memoria, mentre per quanto concerne il secondo malfunzionamento, è dovuto a modifiche eseguite mentre si sta iterando sulla lista. Se si prova a modificare la referenza in volatile, si riesce ad evitare il problema di visibilità della memoria, ma rimane comunque da risolvere il problema delle modifiche concorrenti.

Soluzione con blocchi synchronized

Se si risolve il problema degli accessi concorrenti con dei blocchi synchronized bisogna eseguire tutte le operazioni sulla sharedList usando lo stesso oggetto di sincronizzazione. Approccio parecchio penalizzante dal punto di vista delle performances.

Modifiche nella classe S7Esercizio2:

```
synchronized (S7Esercizio2.sharedPhrase) {
  S7Esercizio2.sharedPhrase.add(getWord());
}
```

Modifiche nella classe ReadWorker:

```
synchronized (S7Esercizio2.sharedPhrase) {
  // Build phrase string from shares words
  final Iterator<String> iterator = S7Esercizio2.sharedPhrase.iterator();
  while (iterator.hasNext()) {
    sb.append(iterator.next());
    sb.append(" ");
  }
}
```

Soluzione con synchronized collection

Introducendo una synchronized collection é possibile rimuovere il blocco synchronized nella classe *S7Esercizio2* in quanto il metodo *add* è già protetto dall'intrinsic lock. Invece, per l'operazione eseguita dal ReadWorker, è necessario mantenere il blocco synchronized (client-side locking). Anche in questo caso le performances sono penalizzate dal blocco synchronized molto grande.

Modifiche nella classe S7Esercizio2:

```
static final List<String> sharedPhrase = Collections.synchronizedList(new ArrayList<String>());
```

Soluzione con concurrent collection

Introducendo infine una *CopyOnWriteArrayList* si può rimuovere anche la sincronizzazione precedentemente necessaria durante l'iterazione sull'array. Però, va tenuto presente che in questa versione, l'iteratore restituito dalla collection lavora su una copia della lista e che quindi la lista potrebbe venire nel frattempo modificata.

Modifiche nella classe S7Esercizio2:

```
static final List<String> sharedPhrase = new CopyOnWriteArrayList<String>();
```

Tabella riassuntiva delle versioni:

	Blocchi synchronized	**Synchronized ArrayList**	**CopyOnWriteArrayList**
tempo simulazione [ms]	12’330	12’027	10’316
modifiche identificate (media)	10.3	10.5	10.1
confronti per readerThread (media)	271’428.5	236’820.9	1’180’135.0

Confrontando i tempi d’esecuzione delle soluzioni, si nota come le prime due abbiano tempi d’esecuzione quasi identici, visto che risolvono il problema di accesso concorrente con tecniche identiche. Invece, la versione che utilizza CopyOnWriteArrayList ha tempi d’esecuzione migliori. Il vantaggio di questa versione è che riesce a percorrere la lista in lettura, senza dover mantenere il lock per tutta l’operazione. Questa caratteristica è resa ancora più evidente osservando il numero dei confronti eseguiti dalle versioni. Lo svantaggio che può capitare è che quest’iterazione esegua su dati non necessariamente attuali della lista. L’iteratore opera su una copia della lista. La lista originale nel frattempo può essere già stata modificata.

Soluzione esercizio 3

Soluzione con ArrayList

Se si usa una ArrayList come casella di posta in entrata è necessario proteggere ogni accesso ad essa con un lock. In questo caso è stato scelto di usare dei blocchi synchronized, usando l'istanza dell'ArrayList come oggetto di sincronizzazione.

```
import java.util.ArrayList;

public class S7Esercizio3 {

  private static class Amico implements Runnable {
    private final String nome;
    private final ArrayList<String> postaEntrata;
    private Amico other;

    public Amico(final String nome) {
      this.nome = nome;
      this.postaEntrata = new ArrayList<String>();
    }

    @Override
    public void run() {
      final Random random = new Random();
      final int nextInt = 1 + random.nextInt(5);
      for (int i = 0; i < nextInt; i++) {
        final String msg = new String("Messaggio" + i + " da " + nome);
        // Mette la lettera nella bucalettere dell'amico
        synchronized (other.postaEntrata) {
          other.postaEntrata.add(msg);
        }
      }

      int lettere = 0;
      while (true) {
        String inMessge;
        do {
          // Controlla la propria bucalettera
          synchronized (postaEntrata) {
            if (postaEntrata.isEmpty())
              inMessge = null;
            else
              inMessge = postaEntrata.remove(0);
          }
        } while (inMessge == null);

        log("Ricevuto " + inMessge);

        if (lettere == 150) {
          log("Ho finito le lettere!");
          break;
        }

        try {
          Thread.sleep(5 + random.nextInt(46));
        } catch (final InterruptedException e) {
          e.printStackTrace();
        }

        final String msg = new String("Risposta" + lettere + " da " + nome);
        // Metti la lettera nella bucalettere dell'amico.
        synchronized (other.postaEntrata) {
          other.postaEntrata.add(msg);
        }
        lettere++;
      }
    }

    public void setAmico(final Amico other) {
      this.other = other;
    }

    private final void log(final String msg) {
      System.out.println(nome + ": " + msg);
```

```
    }
  }

  public static void main(final String[] args) {
    final Amico uno = new Amico("Pippo");
    final Amico due = new Amico("Peppa");

    uno.setAmico(due);
    due.setAmico(uno);

    final Thread tUno = new Thread(uno);
    final Thread tDue = new Thread(due);

    System.out.println("Simulation started!");
    tUno.start();
    tDue.start();
    try {
      tUno.join();
      tDue.join();
    } catch (final InterruptedException e) {
      e.printStackTrace();
    }
    System.out.println("Simulation finished!");
  }
}
```

La soluzione con le concurrent queues sarà tema di una prossima esercitazione.

Soluzione esercizio 4

```
enum Day {
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

class RemoveWorker implements Runnable {
  private final int id;
  private final List<Day> allDays;

  public RemoveWorker(final int ID) {
    this.id = ID;
    this.allDays = new ArrayList<Day>(Arrays.asList(Day.values()));
  }

  @Override
  public void run() {
    final Random random = new Random();
    while (true) {
      // Get random day from available days
      final int nextDayIndex = random.nextInt(allDays.size());
      final Day curDay = allDays.get(nextDayIndex);

      String oldString;
      String newString = null;
      int tries = 0;
      do {
        tries++;

        oldString = S7Esercizio4.allLettersByDay.get(curDay);
        if (oldString.isEmpty()) {
          // Remove empty day from local days list
          allDays.remove(curDay);
          // Quit if there are no more days left
          if (allDays.isEmpty()) {
            log("All days empty finishing!");
            return;
          }
          // exit do-while loop and pick up another day
          break;
        }

        // Create new String by removing first char
        newString = oldString.substring(1);
      } while (!S7Esercizio4.allLettersByDay.replace(curDay, oldString, newString));

      if (tries > 1)
        log("Updated " + curDay + "  after " + tries + " tries");
    }
  }

  private void log(final String msg) {
    System.out.println("RemoveWorker" + id + ": " + msg);
  }
}

public class S7Esercizio4 {
  private static final int NUM_WORKERS = 30;
  private static final int STRING_LEN = 10_000;

  static Map<Day, String> allLettersByDay = new ConcurrentHashMap<>();

  public static void main(final String[] args) {
    final String characters = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890";
    final Random random = new Random();

    for (final Day day : Day.values()) {
      // Create random string
      final StringBuilder sb = new StringBuilder();
      for (int i = 0; i < STRING_LEN; i++) {
        sb.append(characters.charAt(random.nextInt(characters.length())));
      }
      final String randomString = sb.toString();

      // add random string to map for given day
      allLettersByDay.put(day, randomString);
```

```
      // Write starting values
      System.out.println("Init: " + day + "\t-> " + randomString);
    }

    final List<Thread> allThreads = new ArrayList<>();

    for (int i = 0; i < NUM_WORKERS; i++) {
      allThreads.add(new Thread(new RemoveWorker(i)));
    }

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