Solution exercise 1

Synchronized blocks

To reduce the lock time, but still guarantee thread safety, a synchronized block can be added within each *if / else if* block. As a result, the lock operation is performed only when the HashMap is read or modified.

```
while (--cnt > 0) {
  final String key = getClass().getSimpleName() + random.nextInt(A6Exercise1.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");
      }
    }
  }
}
```

Concurrent Collections

Using a ConcurrentHashMap, it is possible to take advantage of the compound actions made available by the ConcurrentMap interface and remove all the synchronization performed by the synchronized blocks. To obtain thread-safety, it is possible to take advantage of the atomicity provided by the *putIfAbset()*, *remove()* and *replace()* methods.

```
private final static ConcurrentHashMap<String, Integer> sharedMap
[...]                                                    = new ConcurrentHashMap<String, Integer>();
while (--cnt > 0) {
  final String key = getClass().getSimpleName() + random.nextInt(A6Exercise1.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");
  }}
```

Solution exercise 2

The program is composed of 15 threads (ReadWorker) and the main thread. All threads access a shared ArrayList (sharedPhrase). The main thread adds, at regular intervals, one of the six strings contained in the array nouns to the ArrayList. Each reading thread on the other hand composes a string with the words contained in the sharedPhrase and compares the produced string with a local copy. When the two strings differ, the local copy is replaced with the new string. In addition, each reading thread keeps track of how many reads it can perform from the shared ArrayList and how many new phrases it can read.

When running the program, it is possible to remark several problems: some ReadWorkers report 0 as a string count, while others manage to count correctly. Also, sometimes ConcurrentModificationExceptions are thrown. The first issue is caused by a memory visibility problem that can be solved by modifying the reference to volatile. The second issue is instead caused by the iteration over the list while other threads make changes to it, which cannot be fixed using volatile.

Solution with synchronized block

To solve the problem of concurrent accesses using synchronized blocks, all operations on the sharedList have to be protected with the same intrinsic lock. This approach is quite heavy from the point of view of performances.

Changes to the A6Exercise2 class:

```
synchronized (A6Eexercise2.sharedPhrase) {
  A6Exercise2.sharedPhrase.add(getWord());
}
```

Changes to the ReadWorker class:

```
synchronized (A6Eexercise2.sharedPhrase) {
  // Build phrase string from shares words
  final Iterator<String> iterator = A6Eexercise2.sharedPhrase.iterator();
  while (iterator.hasNext()) {
    sb.append(iterator.next());
    sb.append(" ");
  }
}
```

Solution with synchronized collection

By introducing a synchronized collection, it is possible to remove the synchronized block in the A6Exercise2 class because the add method is already protected by the intrinsic lock of the synchronized collection. Instead, for the operation performed by the ReadWorker, it is required to maintain the synchronized block (client-side locking). Also, in this case the performances are compromised by the long-lasting synchronized operation.

Changes to A6Exercise2 class:

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

Solution with concurrent collection

Finally, by introducing a CopyOnWriteArrayList, it is possible remove the synchronization previously needed during the iteration on the array. However, remark that the iterator returned by the collection works on a copy of the list and therefore the list could get modified in the meantime it is traversed.

Changes to A6Exercise2 class:

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

Version's recap table:

	Synchronized blocks	**Synchronized ArrayList**	**CopyOnWriteArrayList**
Simulation time[ms]	12'330	12'027	10'316
Identified changes (average)	10.3	10.5	10.1
Comparisons bs reader thread (average)	271'428.5	236'820.9	1'180'135.0

By comparing the execution times of all solutions, it can be remarked how the first two have almost the same execution times. The problem of concurrent access is solved with similar solutions. Instead, when using the CopyOnWriteArrayList the execution time improves. The advantage of the third version is that it can iterate over the list, without maintaining the lock during the whole read operation. This advantage is even more obvious, when looking at the number of comparisons performed by the various versions. The disadvantage is that this iteration executes on data that could get old while traversing the list, because the iterator operates on a copy of the list.

Solution exercise 3

```
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 = A6Exercise3.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 (!A6Exercise3.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 A6Exercise3 {
  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!");
  }
}
```