Soluzione esercizio 1

Il programma soffre di problemi di visibilità della memoria per la variabile condivisa *isRunning*. Se si prova ad eseguire il programma (su un computer con almeno 2 cores), i worker thread restano bloccati nel ciclo `while (!isRunning) {}`, non riuscendo a vedere la modifica di stato della variabile. Visto che la variabile viene modificata solamente dal main thread, mentre tutti gli altri threads si limitano a leggerne il valore, si può risolvere questo problema di visibilità dichiarando la variabile come volatile.

Se si prova ad eseguire nuovamente il programma, ci si rende conto che non riesce a terminare in seguito ad un ulteriore problema di visibilità della memoria legato alla variabile *finished*. La variabile viene letta dal main thread e incrementata da ogni worker thread quando termina l'esecuzione. Mettendo anche questa variabile come volatile, si risolverebbe il problema legato alla visibilità, ma non verrebbe garantita l'atomicità dell'operazione di incremento (l'operazione `finished++` é in realtà una lettura, un incremento ed una scrittura!). Potrebbe cioè accadere che due o più threads riescano a terminare nel medesimo istante, riescano a leggere lo stesso valore e riescano a scrivere lo stesso risultato. Per garantire anche l'atomicità dell'operazione d'incremento si può: racchiudere l'operazione in un blocco synchronized, oppure introdurre dei metodi synchronized che operano sulla variabile *finished,* oppure, più semplicemente, trasformare la variabile da *int* in *AtomicInteger,* sostituendo l'incremento con l'invocazione del metodo *incrementAndGet()* e la lettura con il metodo *get()*.

```
class Worker implements Runnable {
  public static volatile boolean isRunning = false;
  public static AtomicInteger finished = new AtomicInteger(0);

  private int count = 0;
  private final int id;
  private final Random random;

  public Worker(final int id) {
    this.id = id;
    this.random = new Random();
  }

  @Override
  public void run() {
    System.out.println("Worker" + id + " waiting to start");
    while (!isRunning) {
      // Wait!
    }

    System.out.println("Worker" + id + " started");
    for (int i = 0; i < 10; i++) {
      count += random.nextInt(40) + 10;
      try {
        Thread.sleep(random.nextInt(151) + 100);
      } catch (final InterruptedException e) {
        e.printStackTrace();
      }
    }

    System.out.println("Worker" + id + " finished");
    finished.incrementAndGet();
  }

  public void printResult() {
    System.out.println("Worker" + id + " reached " + count);
  }
}

public class S3Esercizio1 {
  public static void main(final String[] args) {

    final List<Worker> allWorkers = new ArrayList<>();
    final List<Thread> allThread = new ArrayList<>();
```

```
    for (int i = 1; i <= 10; i++) {
      final Worker target = new Worker(i);
      allWorkers.add(target);
      final Thread e = new Thread(target);
      allThread.add(e);
      e.start();
    }

    try {
      Thread.sleep(1000);
    } catch (final InterruptedException e) {
    }

    System.out.println("Main thread starting the race!");
    Worker.isRunning = true;

    while (Worker.finished.get() < allWorkers.size()) {
      // Wait
    }

    for (final Worker worker : allWorkers) {
      worker.printResult();
    }

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

Soluzione esercizio 2

Prima versione senza sincronizzazione

```
class Sensore implements Runnable {
  final int soglia;

  public Sensore(final int soglia) {
    this.soglia = soglia;
  }

  @Override
  public void run() {
    System.out.println("Sensore[" + soglia + "]: inizio ad osservare!");
    while (true) {
      if (S3Esercizio2.amount > soglia) {
        S3Esercizio2.amount = 0;
        System.out.println("Sensore[" + soglia + "]: soglia superata!");
        return;
      }
    }
  }
}

public class S3Esercizio2 {
  static int amount = 0;

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

    for (int i = 1; i <= 10; i++) {
      final int soglia_sensore = (i * 10);
      threads.add(new Thread(new Sensore(soglia_sensore)));
    }

    for (final Thread t : threads)
      t.start();

    while (true) {
      final int increment = ThreadLocalRandom.current().nextInt(1, 9);

      S3Esercizio2.amount += increment;
```

```
      System.out.println("Attuatore: ho incrementato lo stato a " + S3Esercizio2.amount);
      if (S3Esercizio2.amount > 120)
          break;

      try {
        Thread.sleep(ThreadLocalRandom.current().nextLong(5, 11));
      } catch (final InterruptedException e) {
        e.printStackTrace();
      }
    }

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

La prima versione contiene solamente la logica richiesta dall'esercizio, senza alcuna protezione dello stato condiviso. Se si esegue il programma su una macchina con più cores, appare evidente il problema di visibilità della memoria. Il main thread incrementa la variabile *amount,* ma nessun thread *Sensore* riesce a vedere gli incrementi di tale variabile. Ogni thread osserva una propria copia dello stato condiviso presente nella cache del processore.

Versione con variabile volatile

```
static volatile int amount = 0;
```

Specificando la variabile *amount* come volatile, si riesce a rimuovere il problema di visibilità. I threads *Sensore* reagiscono all'aumentare della variabile *amount* e terminano la propria esecuzione appena la propria soglia viene superata. Sono però presenti diverse race conditions, per le quali la variabile volatile non riesce a garantire atomicità. La prima è una ReadModifyWrite nel metodo main della classe S3Esercizio2, quando viene fatto l'incremento, con inclusa compound action con la verifica successiva dell'if:

```
S3Esercizio2.amount += increment;
System.out.println("Attuatore: ho incrementato lo stato a " + S3Esercizio2.amount);
if (S3Esercizio2.amount > 120)
```

mentre un'altra race condition di tipo CheckThenAct è presente nella classe Sensore:

```
 if (S3Esercizio2.amount > soglia) {
     S3Esercizio2.amount = 0;
```

Versione con variabile atomica

Specificando la variabile *amount* come AtomicInteger, si può correggere la race condition ReadModifyWrite, sostituendo l'incremento con la chiamata al metodo *addAndGet*. Utilizzando il valore di ritorno del metodo (stack confined), si può inoltre rendere più robusto il programma, sistemando anche la compound action.

```
final int newAmount = S3Esercizio2.amount.addAndGet(increment);
System.out.println("Attuatore: ho incrementato lo stato a "+ newAmount);
if (newAmount > 120)
    break;
```

Per la seconda race condition invece, con le conoscenze attuali, non vi è soluzione con le variabili atomiche.

Versione con ReentrantLocks

Introducendo un ReentrantLock si possono introdurre i metodi *incrementAndGet* e *resetIfAbove*, mentre la variabile *amount* può tornare ad essere di tipo int al posto di un AtomicInteger.

```
private static ReentrantLock lock = new ReentrantLock();
private static int amount = 0;

static int incrementAndGet(final int step) {
  lock.lock();
  try {
    amount += step;
    return amount;
  } finally {
    lock.unlock();
  }
}

static boolean resetIfAbove(final int threshold) {
  lock.lock();
  try {
    if (amount > threshold) {
      amount = 0;
      return true;
    }
    return false;
  } finally {
    lock.unlock();
  }
}
```

Nella classe *Sensore* va introdotto il while loop che continua ad invocare il metodo *resetIfAbove,* in modo che il thread continui a verificare se il valore condiviso ha superato la soglia.

```
while (!S3Esercizio2.resetIfAbove(soglia)) { /* Busy wait */ }
```

Invece, nella classe principale va riscritto il codice per l'incremento nel modo seguente:

```
final int newAmount = S3Esercizio2.incrementAndGet(increment);
System.out.println("Attuatore: ho incrementato lo stato a " + newAmount);
if (newAmount > 120)
    break;
```

Versione con ReentrantReadWriteLock

Infine si può provare a risolvere il problema introducendo un ReentrantReadWriteLock estraendo dal metodo *resetIfAbove* il metodo *isAbove* che sfruttando il readlock verifica se il valore è superiore alla soglia specificata. Il writelock invece viene usato per il reset nel caso la soglia venga superata e per l'incremento da parte del main.
Purtroppo questa soluzione, apparentemente più efficiente, non è corretta in quanto contiene una race condition di tipo check-then-act. Tra l'invocazione di *isAbove* e l'eventuale acquisizione del writelock per il reset il thread potrebbe venir sospeso e nel frattempo cambiare il valore della variabile *amount*.

```
private static ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
private static WriteLock writelock = lock.writeLock();
private static ReadLock readlock = lock.readLock();

private static int amount = 0;

static int incrementAndGet(final int step) {
  writelock.writeLock().lock();
  try {
    amount += step;
    return amount;
  } finally {
    writelock.unlock();
  }
}
```

```
  private static boolean isAbove(final int threshold) {
    readlock.lock();
    try {
      return amount > threshold;
    } finally {
      readlock.readLock().unlock();
    }
  }

  static boolean resetIfAbove(final int threshold) {
    if (isAbove(threshold)) {
      writelock.lock();
      amount = 0;
      writelock.unlock();
      return true;
    }
    return false;
  }
```

Soluzione esercizio 3

```
public class S3Esercizio3 {
  // Lock e shared array: ogni accesso ad sharedArray deve essere protetto usando il lock!
  final static Lock lock = new ReentrantLock();
  final static int sharedArray[] = new int[5];

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

    for (int i = 0; i < 10; i++)
      // Crea worker threads specificando il corpo del runnable tramite lambda expression
      allThreads.add(new Thread(() -> {
        final Random random = new Random<>();
        for (int j = 0; j < 10000; j++) {
          // Scelta della posizione e calcolo del numero da sommare non necessitano del lock!
          final int pos = random.nextInt(sharedArray.length);
          final int num = 10 + random.nextInt(51);
          // Richiesta di accesso esclusivo allo stato condiviso
          lock.lock();
          try {
            sharedArray[pos] += num;
            if (sharedArray[pos] > 500)
              sharedArray[pos] = 0;
          } finally {
            // Fine dell'accesso esclusivo allo stato condiviso
            lock.unlock();
          }

          // Attesa prima della prossima operazione
          try {
            Thread.sleep(ThreadLocalRandom.current().nextLong(2, 6));
          } catch (final Exception e) {
            e.printStackTrace();
          }
        }
      }));

    // Avvia threads
    for (final Thread thread : allThreads) {
      thread.start();
      System.out.println(thread + " avviato");
    }

    // Attesa terminazione dei threads
    for (final Thread thread : allThreads)
      try {
        thread.join();
        System.out.println(thread + " terminato");
      } catch (final InterruptedException e) {
        e.printStackTrace();
      }
  }
}
```