Esercizio 1 - *Soluzione con la tecnica di wait / notify:*

```
class WaitNotifyWorker implements Runnable {
  final int index;
  // Variabili comuni
  static final Object lock = new Object();
  static int numRowsCalculated = 0;
  static int numColsCalculated = 0;

  public WaitNotifyWorker(int index) {
    this.index = index;
  }

  @Override
  public void run() {
    log("inizio a sommare la riga");
    final int result = sumRow(index);

    synchronized (lock) {
      S8Esercizio1WaitNotify.rowSum[index] = result;
      numRowsCalculated++;
      lock.notifyAll();
      log("in attesa che tutte le somme su riga siano terminate");
      try {
        while (numRowsCalculated != 10) {
          lock.wait();
        }
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }

    // Risveglia main
    synchronized (S8Esercizio1WaitNotify.matrix) {
      S8Esercizio1WaitNotify.dataReadyForMain = true;
      S8Esercizio1WaitNotify.matrix.notify();
    }
    log("inizio a sommare la colonna");
    S8Esercizio1WaitNotify.colSum[index] = sumColumn(index);

    synchronized (lock) {
      numColsCalculated++;
      lock.notifyAll();
      log("in attesa che tutte le somme su colonna siano terminate");
      try {
        while (numColsCalculated != 10) {
          lock.wait();
        }
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    log("finito");
  }

  private static final int sumRow(final int row) {
    int result = 0;
    for (int col = 0; col < S8Esercizio1WaitNotify.matrix[row].length; col++)
      result += S8Esercizio1WaitNotify.matrix[row][col];
    return result;
  }

  private static final int sumColumn(final int row) {
    int temp = 0;
    for (int col = 0; col < S8Esercizio1WaitNotify.matrix.length; col++)
      temp += S8Esercizio1WaitNotify.matrix[col][row];
    return temp;
  }

  private final void log(final String msg) {
    System.out.println(getClass().getSimpleName() + "[" + index + "]: " + msg);
  }
}

public class S8Esercizio1WaitNotify {
  final static int[][] matrix = new int[10][10];
```

```
  final static int[] rowSum = new int[matrix.length];
  final static int[] colSum = new int[matrix[0].length];
  static boolean dataReadyForMain = false;

  public static void main(String[] args) {
    // Inizializza matrice con valori random
    initMatrix();
    // Stampa matrice
    System.out.println("Matrice:");
    printMatrix();

    final List<Thread> allThreads = new ArrayList<>();
    for (int i = 0; i < 10; i++)
      allThreads.add(new Thread(new WaitNotifyWorker(i)));

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

    synchronized (matrix) {
      System.out.println("Main in attesa delle somme delle righe");
      while (!dataReadyForMain)
        try {
          matrix.wait();
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
    }
    // Stampa somma delle colonne
    int sommaRighe = 0;
    for (int i = 0; i < rowSum.length; i++)
      sommaRighe += rowSum[i];
    System.out.println("Somme delle righe: " + sommaRighe);

    System.out.println("Main in attesa delle somme delle colonne");
    for (final Thread t : allThreads)
      try {
        t.join();
      } catch (InterruptedException e) {
      }

    // Stampa somma delle colonne
    int sommaColonne = 0;
    for (int i = 0; i < colSum.length; i++)
      sommaColonne += colSum[i];
    System.out.println("Somme delle colonne: " + sommaColonne);
    System.out.println("Finito");
  }

  private static void initMatrix() {
    Random r = new Random();
    for (int row = 0; row < matrix.length; row++) {
      for (int col = 0; col < matrix[row].length; col++) {
        matrix[row][col] = 1 + r.nextInt(100);
      }
    }
  }

  private static void printMatrix() {
    for (int i = 0; i < matrix.length; i++)
      printArray(matrix[i]);
  }

  private static void printArray(final int[] array) {
    for (int i = 0; i < array.length; i++)
      System.out.print(array[i] + "\t");
    System.out.println();
  }
}
```

Esercizio 1 - Soluzione che utilizza le Conditions

```
class ConditionalVariableWorker implements Runnable {
  final int index;
  // Variabili comuni
  static final Lock lock = new ReentrantLock();
  static final Condition stepCompleteCondition = lock.newCondition();
  static final Condition notifyMainCondition = lock.newCondition();
  static boolean dataReadyForMain = false;
  static int numRowsCalculated = 0;
  static int numColsCalculated = 0;

  public ConditionalVariableWorker(final int index) {
    this.index = index;
  }

  @Override
  public void run() {
    log("inizio a sommare la riga");
    S8Esercizio1ConditionVariable.rowSum[index] = sumRow(index);

    try {
      lock.lock();
      numRowsCalculated++;
      stepCompleteCondition.signalAll();
      log("in attesa che tutte le somme su riga siano terminate");
      try {
        while (numRowsCalculated != 10) {
          stepCompleteCondition.await();
        }
      } catch (final InterruptedException e) {
        e.printStackTrace();
      }
    } finally {
      lock.unlock();
    }

    // Risveglia main
    try {
      lock.lock();
      dataReadyForMain = true;
      notifyMainCondition.signal();
    } finally {
      lock.unlock();
    }
    log("inizio a sommare la colonna");
    S8Esercizio1ConditionVariable.colSum[index] = sumColumn(index);

    try {
      lock.lock();
      numColsCalculated++;
      stepCompleteCondition.signalAll();
      log("in attesa che tutte le somme su colonna siano terminate");
      try {
        while (numColsCalculated != 10) {
          stepCompleteCondition.await();
        }
      } catch (final InterruptedException e) {
        e.printStackTrace();
      }
    } finally {
      lock.unlock();
    }
    log("finito");
  }

  private final int sumRow(final int row) {
    int result = 0;
    for (int col = 0; col < S8Esercizio1ConditionVariable.matrix[row].length; col++)
      result += S8Esercizio1ConditionVariable.matrix[row][col];
    return result;
  }

  private final int sumColumn(final int row) {
    int temp = 0;
    for (int col = 0; col < S8Esercizio1ConditionVariable.matrix.length; col++)
```

```
      temp += S8Esercizio1ConditionVariable.matrix[col][row];
    return temp;
  }

  private final void log(final String msg) {
    System.out.println(getClass().getSimpleName() + "[" + index + "]: " + msg);
  }
}

public class S8Esercizio1ConditionVariable {
  final static int[][] matrix = new int[10][10];
  final static int[] rowSum = new int[matrix.length];
  final static int[] colSum = new int[matrix[0].length];

  public static void main(final String[] args) {
    // Inizializza matrice con valori random
    initMatrix();
    // Stampa matrice
    System.out.println("Matrice:");
    printMatrix();

    final List<Thread> allThreads = new ArrayList<>();
    for (int i = 0; i < 10; i++)
      allThreads.add(new Thread(new ConditionalVariableWorker(i)));
    for (final Thread t : allThreads)
      t.start();

    System.out.println("Main in attesa delle somme delle righe");
    ConditionalVariableWorker.lock.lock();
    try {
      while (!ConditionalVariableWorker.dataReadyForMain) {
        try {
          ConditionalVariableWorker.notifyMainCondition.await();
        } catch (final InterruptedException e) {
          e.printStackTrace();
        }
      }
    } finally {
      ConditionalVariableWorker.lock.unlock();
    }

    // Stampa somma delle colonne
    int sommaRighe = 0;
    for (int i = 0; i < rowSum.length; i++)
      sommaRighe += rowSum[i];
    System.out.println("Somme delle righe: " + sommaRighe);

    System.out.println("Main in attesa delle somme delle colonne");
    for (final Thread t : allThreads)
      try {
        t.join();
      } catch (final InterruptedException e) {
      }

    // Stampa somma delle colonne
    int sommaColonne = 0;
    for (int i = 0; i < colSum.length; i++)
      sommaColonne += colSum[i];
    System.out.println("Somme delle colonne: " + sommaColonne);
    System.out.println("Finito");
  }

  private static void initMatrix() {
    final Random r = new Random();
    for (int row = 0; row < matrix.length; row++) {
      for (int col = 0; col < matrix[row].length; col++) {
        matrix[row][col] = 1 + r.nextInt(100);
      }
    }
  }

  private static void printMatrix() {
    for (int i = 0; i < matrix.length; i++)
      printArray(matrix[i]);
  }
```

```
  private static void printArray(final int[] array) {
    for (int i = 0; i < array.length; i++)
      System.out.print(array[i] + "\t");
    System.out.println();
  }
}
```

Esercizio 1 - Soluzione che utilizza i synchronizers

```
class SynchronizerWorker implements Runnable {
  final int index;

  public SynchronizerWorker(final int index) {
    this.index = index;
  }

  @Override
  public void run() {
    log("inizio a sommare la riga");
    S8Esercizio1Synchronizer.rowSum[index] = sumRow(index);
    try {
      log("in attesa che tutte le somme su riga siano terminate");
      S8Esercizio1Synchronizer.barrier.await();
    } catch (final InterruptedException e) {
      log("interrupted!");
      return;
    } catch (final BrokenBarrierException e) {
      log("broken!");
      return;
    }

    log("inizio a sommare la colonna");
    S8Esercizio1Synchronizer.colSum[index] = sumColumn(index);

    log("finito");
  }

  private final int sumRow(final int row) {
    int result = 0;
    for (int col = 0; col < S8Esercizio1Synchronizer.matrix[row].length; col++)
      result += S8Esercizio1Synchronizer.matrix[row][col];
    return result;
  }

  private final int sumColumn(final int row) {
    int temp = 0;
    for (int col = 0; col < S8Esercizio1Synchronizer.matrix.length; col++)
      temp += S8Esercizio1Synchronizer.matrix[col][row];
    return temp;
  }

  private final void log(final String msg) {
    System.out.println(getClass().getSimpleName() + "[" + index + "]: " + msg);
  }
}

public class S8Esercizio1Synchronizer {
  // Variabili comuni
  final static CyclicBarrier barrier = new CyclicBarrier(11);
  final static int[][] matrix = new int[10][10];
  final static int[] rowSum = new int[matrix.length];
  final static int[] colSum = new int[matrix[0].length];

  public static void main(final String[] args) {
    // Inizializza matrice con valori random
    initMatrix();
    // Stampa matrice
    System.out.println("Matrice:");
    printMatrix();

    final List<Thread> allThreads = new ArrayList<>();
    for (int i = 0; i < 10; i++)
      allThreads.add(new Thread(new SynchronizerWorker(i)));
    for (final Thread t : allThreads)
      t.start();
```

```
    try {
      barrier.await();
    } catch (final InterruptedException e1) {
      e1.printStackTrace();
    } catch (final BrokenBarrierException e1) {
      e1.printStackTrace();
    }

    // Stampa somma delle colonne
    int sommaRighe = 0;
    for (int i = 0; i < rowSum.length; i++)
      sommaRighe += rowSum[i];
    System.out.println("Somme delle righe: " + sommaRighe);

    System.out.println("Main in attesa delle somme delle colonne");
    for (final Thread t : allThreads)
      try {
        t.join();
      } catch (final InterruptedException e) {
      }

    // Stampa somma delle colonne
    int sommaColonne = 0;
    for (int i = 0; i < colSum.length; i++)
      sommaColonne += colSum[i];
    System.out.println("Somme delle colonne: " + sommaColonne);

    System.out.println("Finito");
  }

  private static void initMatrix() {
    final Random r = new Random();
    for (int row = 0; row < matrix.length; row++) {
      for (int col = 0; col < matrix[row].length; col++) {
        matrix[row][col] = 1 + r.nextInt(100);
      }
    }
  }

  private static void printMatrix() {
    for (int i = 0; i < matrix.length; i++)
      printArray(matrix[i]);
  }

  private static void printArray(final int[] array) {
    for (int i = 0; i < array.length; i++)
      System.out.print(array[i] + "\t");
    System.out.println();
  }
}
```

Esercizio 2 - Soluzione con ConcurrentLinkedQueue

```
public class S8Esercizio2ConcurrentLinkedQueue {

  private static class Amico implements Runnable {
    private final String nome;
    private final ConcurrentLinkedQueue<String> postaEntrata = new ConcurrentLinkedQueue<String>();
    private Amico other;

    public Amico(final String nome) {
      this.nome = nome;
    }

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

      int lettere = 0;
      while (true) {
        String inMessge;
        do {
          // Controlla posta in entrata
          inMessge = postaEntrata.poll();
        } 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);
        // Mette la lettera nella bucalettere dell'amico
        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);

    tUno.start();
    tDue.start();
    try {
      tUno.join();
      tDue.join();
    } catch (final InterruptedException e) { e.printStackTrace(); }
  }
}
```

Esercizio 2 - Soluzione con LinkedBlockingQueue

```
public class S8Esercizio2LinkedBlockingQueue {

  private static class Amico implements Runnable {
    private final String nome;
    private final BlockingQueue<String> postaEntrata = new LinkedBlockingQueue<String>();
    private Amico other;

    public Amico(final String nome) {
      this.nome = nome;
    }

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

      int lettere = 0;
      while (true) {
        String inMessge;
        try {
          // Controlla posta in entrata
          inMessge = postaEntrata.take();
        } catch (final InterruptedException e) { /* Termina se interrotto! */ break; }

        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);
        // Mette la lettera nella bucalettere dell'amico
        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);

    tUno.start();
    tDue.start();
    try {
      tUno.join();
      tDue.join();
    } catch (final InterruptedException e) { e.printStackTrace(); }
  }
}
```

Esercizio 3

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.atomic.AtomicBoolean;

final class S8Es3Timer {
  private long start = -1;
  private long stop = -1;

  final public void start() {
    start = System.currentTimeMillis();
  }

  final public void stop() {
    stop = System.currentTimeMillis();
  }

  final public long getElapsed() {
    return (start < 0 || stop < 0) ? 0 : stop - start;
  }
}

final class Testimone {
  final int id;

  public Testimone(final int id) {
    this.id = id;
  }

  @Override
  public String toString() {
    return "t" + id;
  }
}

class Corridore implements Runnable {
  private final int id;
  private final Squadra squadra;
  private Testimone testimone;
  private Corridore prossimo;

  public Corridore(final int id, final Squadra squadra, final Testimone testimone) {
    this.id = id;
    this.squadra = squadra;
    this.testimone = testimone;
    this.prossimo = null;
    log("creato");
  }

  public Corridore(final int id, final Squadra squadra) {
    this(id, squadra, null);
  }

  @Override
  public void run() {
    if (testimone != null)
      attendiPartenza();
    else
      attendiTestimone();

    corri();

    if (prossimo != null) {
      passaTestimone();
    } else {
      completaCorsa();
    }
  }

  final private void completaCorsa() {
    squadra.stopTime();
    // Solo il primo riesce a sostituire il true con il false!
```

```
    if (S7Esercizio3.isPrimoAllArrivo.compareAndSet(true, false))
      log("VITTORIA !!");
    else
      log("Perso.");
  }

  final private void corri() {
    final Random random = Random();
    final long millis = 100 + random.nextInt(50);
    log("corro per " + millis + " ms");
    try {
      Thread.sleep(millis);
    } catch (final InterruptedException e) {
      /* do nothing */
    }
  }

  final private void attendiTestimone() {
    log("in attesa del testimone!");
    synchronized (this) {
      while (this.testimone == null) {
        try {
          wait();
        } catch (final InterruptedException e) {
          /* do nothing */
        }
      }
    }
    log("testimone ricevuto!");
  }

  final private void attendiPartenza() {
    log("in attesa del segnale di partenza!");
    S7Esercizio3.partenza.countDown();
    try {
      S7Esercizio3.partenza.await();
    } catch (final InterruptedException e) {
      /* do nothing */
    }
    squadra.startTime();
  }

  final private void passaTestimone() {
    log("passo testimone " + testimone + " a " + prossimo);

    synchronized (prossimo) {
      prossimo.setTestimone(testimone);
      prossimo.notify();
    }
    testimone = null;
  }

  final private void log(final String message) {
    System.out.println(toString() + ": " + message);
  }

  public void setProssimo(final Corridore c) {
    prossimo = c;
  }

  public void setTestimone(final Testimone t) {
    this.testimone = t;
  }

  @Override
  public String toString() {
    return getClass().getSimpleName() + id + "_" + squadra;
  }
}

class Squadra {
  final private int id;
  final private S8Es3Timer time;
  final private ArrayList<Corridore> corridori;

  public Squadra(final int id, final int numCorridori) {
```

```
    this.id = id;
    this.time = new S8Es3Timer();
    this.corridori = new ArrayList<Corridore>();

    Corridore attuale = null;
    Corridore precedente = new Corridore(0, this, new Testimone(id));
    corridori.add(precedente);

    for (int i = 1; i < numCorridori; i++) {
      attuale = new Corridore(i, this);
      corridori.add(attuale);
      precedente.setProssimo(attuale);
      precedente = attuale;
    }
  }

  public void startTime() {
    time.start();
  }

  public void stopTime() {
    time.stop();
  }

  final public long getElapsed() {
    return time.getElapsed();
  }

  public void logElapsedTime() {
    System.out.println(this + " elapsedTime : " + time.getElapsed() + " ms");
  }

  public ArrayList<Corridore> getAllCorridori() {
    return corridori;
  }

  @Override
  public String toString() {
    return getClass().getSimpleName() + id;
  }
}

public class S8Esercizio3 {
  private final static int NUM_SQUADRE = 4;
  private final static int NUM_CORRIDORI = 10;
  static CountDownLatch partenza = new CountDownLatch(NUM_SQUADRE + 1);
  static AtomicBoolean isPrimoAllArrivo = new AtomicBoolean(true);

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

    // Crea squadre, ogni squadra crea i propri corridori
    for (int i = 0; i < NUM_SQUADRE; i++)
      squadre.add(new Squadra(i, NUM_CORRIDORI));

    // Per ogni squadra, per ogni corridore della squadra, crea thread
    squadre.forEach(squadra -> squadra.getAllCorridori().forEach(c -> threads.add(new Thread(c))));

    System.out.println("Simulation started");
    System.out.println("-----------------------------------");
    // Fa partire tutte le threads
    threads.forEach(Thread::start);

    try {
      Thread.sleep(1000);
      System.out.print("Pronti ...");
      Thread.sleep(1000);
      System.out.print("Partenza...");
      Thread.sleep(1000);
      System.out.println("Via!");
    } catch (final InterruptedException e1) {
      // Do nothing
    }

    // Fa partire la corsa
```

```
    S8Esercizio3.partenza.countDown();

    // Attende che tutte le threads terminano
    for (final Thread t : threads)
      try {
        t.join();
      } catch (final InterruptedException e) {
        /* do nothing */
      }
    System.out.println("-----------------------------------");
    System.out.println("Classifica finale:");

    // Ordina le squadre in funzione dei tempi
    squadre.sort((s1, s2) -> Long.compare(s1.getElapsed(), s2.getElapsed()));

    // stampa tempi
    squadre.forEach(Squadra::logElapsedTime);
    System.out.println("Simulation finished");
  }
}
```

Esercizio 4 - Soluzione con locks espliciti e conditional variables

```
class FantinoCondition implements Runnable {
  private final int id;
  private final Partenza lineaDiPartenza;

  public FantinoCondition(final int id, final Partenza partenza) {
    this.id = id;
    this.lineaDiPartenza = partenza;
  }

  @Override
  public void run() {
    try {
      Thread.sleep(ThreadLocalRandom.current().nextLong(1000, 1050));
    } catch (final Exception e1) { /* non gestita */ }

    System.out.println("Fantino" + id + ": arrivo alla linea di partenza");
    lineaDiPartenza.arriva();
    final long start = System.currentTimeMillis();
    try {
      lineaDiPartenza.attendiPartenza();
    } catch (final Exception e) { /* non gestita */ }

System.out.println("Fantino" + id + ": ha atteso " + (System.currentTimeMillis() - start) + " ms");
  }
}

class Partenza {
  private final Lock lock = new ReentrantLock();
  private final Condition tuttiArrivati = lock.newCondition();
  private int numPartecipanti;

  public Partenza(final int numPartecipanti) {
    this.numPartecipanti = numPartecipanti;
  }

  public void arriva() {
    lock.lock();
    try {
      numPartecipanti--;
      if (numPartecipanti == 0)
        tuttiArrivati.signalAll();
    } finally {
      lock.unlock();
    }
  }

  public void attendiPartenza() throws InterruptedException {
    lock.lock();
    try {
      while (numPartecipanti > 0)
        tuttiArrivati.await();
    } finally {
      lock.unlock();
    }
  }
}

public class S8Esercizio4Conditions {
  public static void main(final String[] args) {
    final Partenza lineaDiPartenza = new Partenza(10);
    final List<Thread> allThreads = new ArrayList<>();

    for (int i = 0; i < 10; i++)
      allThreads.add(new Thread(new FantinoCondition(i, lineaDiPartenza)));

    allThreads.forEach(Thread::start);
    for (final Thread t : allThreads)
      try {
        t.join();
      } catch (final InterruptedException e) { /* non gestita */ }
  }
}
```

Esercizio 4 - Soluzione con synchronizer di tipo CountDownLatch

```
class FantinoSynchronizer implements Runnable {
  private final int id;
  private final CountDownLatch lineaDiPartenza;

  public FantinoSynchronizer(final int id, final CountDownLatch latch) {
    this.id = id;
    this.lineaDiPartenza = latch;
  }

  @Override
  public void run() {
    try {
      Thread.sleep(ThreadLocalRandom.current().nextLong(1000, 1050));
    } catch (final Exception e1) { /* non gestita */ }

    System.out.println("Fantino" + id + ": arrivo alla linea di partenza");
    lineaDiPartenza.countDown();
    final long start = System.currentTimeMillis();
    try {
      lineaDiPartenza.await();
    } catch (final Exception e) { /* non gestita */ }

System.out.println("Fantino" + id + ": ha atteso " + (System.currentTimeMillis() - start) + " ms");
  }
}

public class S8Esercizio4Synchronizers {
  public static void main(final String[] args) {
    final CountDownLatch lineaDiPartenza = new CountDownLatch(10);
    final List<Thread> allThreads = new ArrayList<>();

    for (int i = 0; i < 10; i++)
      allThreads.add(new Thread(new FantinoSynchronizer(i, lineaDiPartenza)));

    allThreads.forEach(Thread::start);
    for (final Thread t : allThreads)
      try {
        t.join();
      } catch (final InterruptedException e) {
        /* non gestita */
      }
  }
}
```