Esercizio 1

Questo esercizio è simile al problema visto in classe.

Problemi:

I membri statici della classe Worker vengono letti e scritti in maniera concorrenziale dai vari thread.

```
public static boolean isRunning = false;
public static int finished = 0;
```

Nel primo caso abbiamo un problema di visibilità (il main thread modifica il valore ma i thread Worker non se ne accorgono).

Nel secondo caso abbiamo un problema di visibilità (i thread Worker modificano i valori e il main thread non se ne accorge) e di "read-modify-write" race condition (i thread Worker possono modificare il valore in concorrenza).

Soluzione:

Rendere volatile "isRunning" e atomica "finished".

```
public static volatile boolean isRunning = false;
public static AtomicInteger finished = new AtomicInteger(0);
```

Esercizio 2

Le classi `CounterNoSync, CounterVolatile, CounterAtomic, CounterExplicitLock, CounterReadWriteLock` implementano l'interfaccia **Counter**

Il main crea un oggetto Counter che viene condiviso dai vari Thread Sensor.

Tipo	Comportamento
`CounterNoSync`	Il Main Thread termina mentre i Thread Sensor continuano a girare. Problema di visibilità.
`CounterVolatile`	Il programma sembra comportarsi bene ma esistono 2 problemi di Race Condition (più evidenti se si aumenta il numero di sensori). Il metodo **add** esegue un'operazione non atomica **"read/modify/write"**. Il metodo **resetAboveThreshold** soffre del problema **"check-then-act".**
`CounterAtomic`	Il programma si comporta bene ed ha anche le **migliori prestazioni.**
`CounterExplicitLock`	Il programma si comporta bene.
`CounterReadWriteLock`	Il programma si comporta bene, non migliora le prestazioni rispetto all'explicit lock.

Test [serie03.CounterAtomic] completed after 313ms . Final counter: 121

Test [serie03.CounterExplicitLock] completed after 1679ms . Final counter: 124

Test [serie03.CounterReadWriteLock] completed after 1564ms . Final counter: 123