Esercizio 1

Questo esercizio simula un sistema autostradale con pedaggi. Le classi presenti sono:

- **S2Esercizio1:** classe principale contenente il main.
- **Autostrada:** espone 3 membri pubblici (nessun metodo).
- **Automobilista:** ha una dipendenza su Autostrada, implementa l'interfaccia Runnable.

Alla partenza del main vengono creati 10 automobilisti, ognuno dei quali viene associato a un **Thread**.

I 10 Thread vengono eseguiti in parallelo mentre il Main Thread aspetta la loro fine per poter poi stampare le statistiche di ogni automobilista.

I 10 automobilisti hanno tutti un riferimento ad un unico oggetto condiviso Autostrada.

Nel metodo run di Automobilista, un loop di 500 iterazioni simula un viaggio con entrate e uscite dall'autostrada. Ogni iterazione avviene in tempi diversi, dovuta al valore random passato a 2 chiamate di Thread.sleep.

Problema:

I valori dell'oggetto Autostrada vengono modificati ad ogni iterazione, ma essendo Autostrada un oggetto condiviso avviene un problema di Race Condition.

```
autostrada.entrate++;
autostrada.uscite++;
autostrada.pedaggi += pedaggioTratta;
```

Soluzione:

I 3 incrementi sono stati rifattorizzati come metodi della classe Autostrada.

```
public void incrEntrate(){ entrate++; }
public void incrUscite(){ uscite++; }
public void addPedaggio(int pedaggio) { pedaggi += pedaggio; }
```

La modifica dei 3 membri dell'oggetto autostrada avviene in modo concorrenziale, ma ogni valore è indipendente dall'altro.

Creo 3 nuove classi **AutostradaSyncBlock**, **AutostradaSyncMethod** e **AutostradaExplicit** che fannno override di questi 3 metodi e proteggono la modifica dei valori in modo **Thread Safe**.

In questo esercizio non ci sono altri punti di accesso ai 3 membri da proteggere, visto che la lettura finale avviene nel Main thread solo dopo il join degli altri thread.

Esercizio 2

Questo esercizio simula l'utilizzo di un bagno pubblico da parte di 10 utenti. Le classi presenti sono:

- **S2Esercizio2:** classe principale contenente il main.
- **BagnoPubblico:** espone 1 metodo pubblico **occupa()** per utilizzare uno degli n=2 bagni. Il metodo torna false se tutti i bagni sono occupati, true se è disponibile almeno un bagno. La classe tiene traccia della disponibilità dei bagni liberi, del numero di utilizzi e del numero di risposte negative. Il tempo di utilizzo varia ogni volta tramite un valore random passato alla Thread.sleep
- **User:** ha una dipendenza su BagnoPubblico, implementa l'interfaccia Runnable.

Alla partenza del main vengono creati 10 utenti, ognuno dei quali viene associato a un **Thread**.

I 10 Thread vengono eseguiti in parallelo mentre il Main Thread aspetta la loro fine per poter poi stampare le statistiche del bagno pubblico e di ogni utente.

I 10 utenti hanno tutti un riferimento ad un unico oggetto condiviso BagnoPubblico.

Nel metodo run di User, un loop di 250 iterazioni simula la richiesta da parte di un utente dell'utilizzo del bagno tramite il metodo occupa(). La classe tiene traccia del numero di volte in cui l'utente è riuscito a usare il bagno e del numero di volte in cui non è riuscito. Ogni iterazione viene ritardata di un tempo random.

Problema:

Ogni Thread esegue il metodo occupa() in maniera concorrenziale. Ma la chiamata a questo metodo ha un side effect sull'oggetto bagno che è condiviso tra i thread. Anche qui c'è un problema di Race Condition.

Soluzione:

Rifattorizzo il codice dentro il metodo occupa() introducendo 2 nuovi metodi:

```
protected boolean occupaSeLibero() {
   // Verifica disponibilita bagni liberi!
   if (occupati < disponibili) {
      // Bagno libero! Occupa
      occupati++;
      totUtilizzi++;
      return true;
   }
   // Tutti i bagni sono occupati!
   totOccupati++;
   return false;
}

protected void libera() {
    occupati--;
}
```

Creo 3 nuove classi **BagnoPubblicoSyncBlock**, **BagnoPubblicoSyncMethod** e **BagnoPubblicoExplicit** che fannno override di questi 2 metodi e proteggono la modifica/lettura dei valori in modo **Thread Safe**.