Soluzione esercizio 1

Il programma è composto dal main thread, che si occupa di aggiornare una variabile condivisa (*sharedValue*) e da 10 threads (*Readers*) che aggiornano un proprio contatore (*localValue*), confrontandolo con la variabile condivisa. Il lock in questo programma è sfruttato dal main thread per aggiornare e garantire corretta visibilità del valore condiviso, mentre viene usato dai *reader* threads per proteggere la compound action di aggiornamento del proprio contatore. La compound action è infatti composta da 3 distinte letture dello stato condiviso: la prima per il confronto, la seconda per l'assegnazione e la terza per la scrittura sull'outputstream. Togliendo il lock è necessario garantire che il funzionamento del programma non cambi! Per garantire atomicità e corretta visibilità dello stato condiviso è possibile sfruttare le caratteristiche delle variabili volatile, dichiarando *sharedValue* come volatile. Inoltre, va risolto il problema delle letture multiple allo stato condiviso da parte dei Reader threads: è infatti possibile che il valore cambi tra una lettura e quella successiva. Introducendo la variabile locale *localSharedValue* si può confinare il valore di sharedValue allo stack di ogni *Reader* thread. Viene eseguita una sola lettura allo stato condiviso durante l'assegnazione di *localSharedValue*, mentre le tre seguenti letture vengono eseguite sulla variabile locale che non può essere modificata da altri threads.

Rispetto alla versione originale, il programma non necessita di alcuna forma di lock, grazie alle proprietà delle variabili volatile e all'utilizzo della tecnica di stack confinement, permettendo ai reader thread di accedere una sola volta in lettura allo stato condiviso.

```
class Reader implements Runnable {
  private final int id;
  private int localValue;

  public Reader(final int id) {
    this.id = id;
    this.localValue = -1;
  }

  @Override
  public void run() {
    while (S5Esercizio1.isRunning.get()) {
      // Read shared value to local variable
      final int localSharedValue = S5Esercizio1.sharedValue;
      // Use local value for updates and compare
      if (localValue != localSharedValue) {
        localValue = localSharedValue;
      } else
        System.out.println("Reader" + id + ": (" + localValue + " == " + localSharedValue + ")");
    }
  }
}

public class S5Esercizio1 {
  final static AtomicBoolean isRunning = new AtomicBoolean(true);
  static volatile int sharedValue = 0;

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

    final Random random = new Random();

    // Create threads
    for (int i = 0; i < 10; i++)
      allThread.add(new Thread(new Reader(i)));
    // Start all threads
    for (final Thread t : allThread)
      t.start();

    for (int i = 0; i < 1000; i++) {
      // 1 Writer Many readers: no need to synchronize
```

```
      S5Esercizio1.sharedValue = random.nextInt(10);
      // Wait 1 ms between updates
      try {
        Thread.sleep(1);
      } catch (final InterruptedException e) {
        e.printStackTrace();
      }
    }

    // Notify all workers that processing has finished
    isRunning.set(false);

    // Wait for all threads to complete
    for (final Thread t : allThread)
      try {
        t.join();
      } catch (final InterruptedException e) {
        e.printStackTrace();
      }
    System.out.println("Simulation terminated.");
  }
}
```

Soluzione esercizio 2

Il programma soffre di un problema di escape della "this" reference in fase di costruzione della classe EventListener. A causa dell'escape, gli oggetti di tipo EventListener vengono pubblicati mentre è ancora in esecuzione il costruttore. Questa situazione può rivelarsi problematica, se il thread dell'EventSource riesce ad accedere ai campi degli oggetti mentre non sono ancora correttamente inizializzati.

Per correggere l'errore è necessario rifattorizzare le classi, eliminando il parametro di tipo EventSource dal costruttore della classe EventListener. La fase di registrazione del listener va posticipata, in modo che sia successiva alla creazione degli oggetti. Così facendo, l'escape della "this" reference non può più avvenire e la costruzione degli oggetti risulta essere sicura.

Il codice è da modificarsi nella seguente maniera:

```
class EventListener {
  protected final int id;

  public EventListener(final int id) {
    // Sleep che facilita l'apparizione del problema. In una situazione
    // reale qui potrebbe fare altre inizializzazioni.
    try {
      Thread.sleep(4);
    } catch (final InterruptedException e) {
      // Thread interrupted
    }

    this.id = id;
  }

  public void onEvent(final int listenerID, final Event e) {
    if (listenerID != id)
      System.out.println("Inconsistent listener ID" + listenerID + " : " + e);
  }
}

public class S5Esercizio2 {
  public static void main(final String[] args) {
    final EventSource eventSource = new EventSource();
    final Thread eventSourceThread = new Thread(eventSource);

    // Fa partire il thread
    eventSourceThread.start();

    // Crea e registra il listener alla sorgente
    final List<EventListener> allListeners = new ArrayList<>();
    for (int i = 0; i < 20; i++) {
      final EventListener listener = new EventListener(i);
      eventSource.registerListener(i, listener);
      allListeners.add(listener);
    }

    // Attendi che il Thread termini
    try {
      eventSourceThread.join();
    } catch (final InterruptedException e) {
      // Thread interrotto
    }
  }
}
```

Soluzione esercizio 3

Il programma presenta un problema di safe-publication. Sia per gli oggetti non thread-safe di tipo SharedState, sia per oggetti thread-safe di tipo ThreadSafeSharedState è necessario eseguire una pubblicazione sicura. La referenza all'oggetto e lo stato dell'oggetto devono essere resi visibili contemporaneamente. Nel programma, durante la pubblicazione dell'oggetto di tipo *IState* non viene utilizzato nessuno strumento di sincronizzazione (come ad esempio i blocchi synchronized o i ReentrantLocks), quindi, il thread in lettura non riesce a vedere correttamente la referenza all'oggetto ed attende all'infinito nel while loop. Il thread che si occupa di costruire l'oggetto, pubblica la referenza, che però risulta visibile solo all'interno della cache del core utilizzato dal thread. Per poter correggere il problema è necessario condividere la referenza con pubblicazione sicura.

Soluzione volatile reference per ThreadSafeSharedState

Specificando la referenza condivisa `S5Esercizio2.sharedState` come volatile, viene garantita corretta visibilità della referenza verso tutti i threads.

```
public class S5Esercizio3 {
	static volatile IState sharedState = null;
...
```

Soluzione synchronized method per ThreadSafeSharedState

Si può risolvere lo stesso problema, introducendo i metodi synchronized *getSharedState()* e *setSharedState()* e sostituendo tutti gli accessi allo stato condiviso con chiamate ai nuovi metodi. La classe *Starter* pubblicherà l'oggetto tramite *setSharedState()*, mentre la classe *Helper* dovrà accedere allo stato condiviso tramite il metodo *getSharedState()*.

```
class Helper implements Runnable {
  @Override
  public void run() {
    System.out.println("Helper : started and waiting until shared state is set!");
    while (true) {
      if (S5Esercizio3.getSharedState() != null)
        break;
    }

    int lastValue = S5Esercizio3.getSharedState().getValue();
    System.out.println("Helper : shared state initialized and current value is " + lastValue
                    + ". Waiting until value changes");

    // Wait until value changes
    while (true) {
      if (lastValue != S5Esercizio3.getSharedState().getValue()) {
        lastValue = S5Esercizio3.getSharedState().getValue();
        break;
      }
    }
    System.out.println("Helper : value changed to " + lastValue + "!");

    for (int i = 0; i < 5000; i++) {
      S5Esercizio3.getSharedState().increment();
      if ((i % 100) == 0)
        try {
          Thread.sleep(1);
        } catch (final InterruptedException e) {
      }
    }
    System.out.println("Helper : completed");
  }
}

class Starter implements Runnable {
  @Override
```

```
  public void run() {
    System.out.println("Starter : sleeping");
    try {
      Thread.sleep(1000);
    } catch (final InterruptedException e) { /* UNHANDLED EXCEPTION */ }

    System.out.println("Starter : initialized shared state");
    // Choose which share to instantiate
    if (S5Esercizio3.THREADSAFE_SHARE)
      S5Esercizio3.setSharedState(new ThreadSafeSharedState());
    else
      S5Esercizio3.setSharedState(new SharedState());

    try {
      Thread.sleep(1000);
    } catch (final InterruptedException e) { /* UNHANDLED EXCEPTION */ }

    for (int i = 0; i < 5000; i++) {
      S5Esercizio3.getSharedState().increment();
      if ((i % 100) == 0)
        try {
          Thread.sleep(1);
        } catch (final InterruptedException e) {
      }
    }
    System.out.println("Starter : completed");
  }
}

public class S5Esercizio3 {
  static IState sharedState = null;

  static synchronized IState getSharedState() {
    return sharedState;
  }

  static synchronized void setSharedState(final IState state) {
    sharedState = state;
  }
…
```

Quando si utilizzano gli oggetti non thread-safe (SharedState) è inoltre necessario proteggere gli accessi concorrenti allo stato condiviso. La soluzione con referenza volatile garantisce infatti solo corretta visibilità alla referenza ma non ha alcun influenza sulla thread-safety dell'oggetto referenziato. Lo stesso vale anche per la precedente soluzione con metodi synchronized. I metodi getSharedState e setSharedState si limitano a controllare / proteggere gli accessi alla referenza e non allo stato dell'oggetto referenziato.

Soluzione volatile reference per SharedState

Per poter garantire Thread Safety allo stato condiviso si può usare direttamente l'istanza dello sharedState come mutex.

```
public class S5Esercizio3 {
       static volatile IState sharedState = null;
...
class Helper implements Runnable {
  @Override
  public void run() {
    System.out.println("Helper : started and waiting until shared state is set!");
    while (true) {
      if (S5Esercizio3.sharedState != null)
        break;
    }

    int lastValue;
    synchronized (S5Esercizio3.sharedState) {
       lastValue = S5Esercizio3.sharedState.getValue();
    }

    System.out.println("Helper : shared state initialized and current value is " + lastValue
```

```
                    + ". Waiting until value changes");
    // Wait until value changes
    while (true) {
      final int curValue;
      synchronized (S5Esercizio3.sharedState) {
        curValue = S5Esercizio3.sharedState.getValue();
      }
      if (lastValue != curValue) {
        lastValue = curValue;
        break;
      }
    }
    System.out.println("Helper : value changed to " + lastValue + "!");

    for (int i = 0; i < 5000; i++) {
      synchronized (S5Esercizio3.sharedState) {
        S5Esercizio3.sharedState.increment();
      }
      if ((i % 100) == 0)
        try {
          Thread.sleep(1);
        } catch (final InterruptedException e) {
      }
    }
    System.out.println("Helper : completed");
  }
}

class Starter implements Runnable {
  @Override
  public void run() {
    System.out.println("Starter : sleeping");
    try {
      Thread.sleep(1000);
    } catch (final InterruptedException e) { /* UNHANDLED EXCEPTION */ }

    // Choose which share to instantiate
    if (S5Esercizio3.THREADSAFE_SHARE)
      S5Esercizio3.sharedState = new ThreadSafeSharedState();
    else
      S5Esercizio3.sharedState = new SharedState();

    try {
      Thread.sleep(1000);
    } catch (final InterruptedException e) { /* UNHANDLED EXCEPTION */ }

    for (int i = 0; i < 5000; i++) {
      synchronized (S5Esercizio3.sharedState) {
        S5Esercizio3.sharedState.increment();
      }
      if ((i % 100) == 0)
        try {
          Thread.sleep(1);
        } catch (final InterruptedException e) {
      }
    }
    System.out.println("Starter : completed");
  }
}
```

Soluzione synchronized method per SharedState

Per poter proteggere gli accessi di modifica allo stato condiviso SharedState si salva in una variabile la referenza ottenuta attraverso il metodo *getSharedState()* e si usa l'istanza stessa come mutex.

```
class Helper implements Runnable {
  @Override
  public void run() {
    System.out.println("Helper : started and waiting until shared state is set!");
    while (true) {
      if (S5Esercizio3.getSharedState() != null)
        break;
    }
```

```
    int lastValue;
    final IState currentState = S5Esercizio3.getSharedState();
    synchronized (currentState) {
      lastValue = currentState.getValue();
    }
    System.out.println("Helper : shared state initialized and current value is " + lastValue
                       + ". Waiting until value changes");

    // Wait until value changes
    while (true) {
      final int curValue;
      final IState curState = S5Esercizio3.getSharedState();
      synchronized (curState) {
        curValue = curState.getValue();
      }
      if (lastValue != curValue) {
        lastValue = curValue;
        break;
      }
    }
    System.out.println("Helper : value changed to " + lastValue + "!");

    for (int i = 0; i < 5000; i++) {
      final IState curState = S5Esercizio3.getSharedState();
      synchronized (curState) {
        curState.increment();
      }
      if ((i % 100) == 0)
        try {
          Thread.sleep(1);
        } catch (final InterruptedException e) {
      }
    }
    System.out.println("Helper : completed");
  }
}

class Starter implements Runnable {
  @Override
  public void run() {
    System.out.println("Starter : sleeping");
    try {
      Thread.sleep(1000);
    } catch (final InterruptedException e) { /* UNHANDLED EXCEPTION */ }

    System.out.println("Starter : initialized shared state");
    // Choose which share to instantiate
    if (S5Esercizio3.THREADSAFE_SHARE)
      S5Esercizio3.setSharedState(new ThreadSafeSharedState());
    else
      S5Esercizio3.setSharedState(new SharedState());

    try {
      Thread.sleep(1000);
    } catch (final InterruptedException e) { /* UNHANDLED EXCEPTION */ }

    for (int i = 0; i < 5000; i++) {
      final IState curState = S5Esercizio3.getSharedState();
      synchronized (curState) {
        curState.increment();
      }
      if ((i % 100) == 0)
        try {
          Thread.sleep(1);
        } catch (final InterruptedException e) {
      }
    }
    System.out.println("Starter : completed");
  }
}
```

Soluzione "Holder degli oggetti immutable"

Per questa versione bisogna introdurre una classe immutable (*SharedStateImmutable*) che contenga la variabile *value* e la classe Holder che invece implementa l'interfaccia IState e che implementi i metodi getValue e increment in modo consistente. Nell'esempio di soluzione proposto, viene fornita un'implementazione thread safe che sfrutta i ReadWriteLocks.

Per verificare il corretto funzionamento, si può sostituire la versione che istanzia ThreadSafeSharedState con ImmutableSharedState.

```
final class Holder {
  private final int value;

  public Holder(int v) {
    this.value = v;
  }

  public int getValue() {
    return value;
  }

  public Holder increment() {
    return new Holder(value + 1);
  }
}

class ImmutableSharedState implements IState {
  private final ReadWriteLock rwlock = new ReentrantReadWriteLock();
  private Holder state = new Holder(0);

  @Override
  public void increment() {
    rwlock.writeLock().lock();
    try {
      state = state.increment();
    } finally {
      rwlock.writeLock().unlock();
    }
  }

  @Override
  public int getValue() {
    rwlock.readLock().lock();
    try {
      return state.getValue();
    } finally {
      rwlock.readLock().unlock();
    }
  }
}
```