Solution exercise 1

The program is composed of the main thread, which updates a shared variable (sharedValue), and ten threads (Readers) which update their own counter (localValue), by comparing it with the shared variable. The main thread uses the explicit lock to perform the update, that guarantee the correct visibility of the shared value. The reader threads use the explicit lock to protect the compound action of updating their own counter. The compound action is composed of 3 read operations of the shared state: the first for comparison, the second for assignment, and the third for writing to the outputstream. By removing the explicit lock, it is required to guarantee that the program's behavior does not change! To guarantee atomicity and correct visibility of the shared state it is possible to exploit the characteristics of volatile variables, declaring sharedValue as volatile. In addition, the problem of multiple read operations of the shared state by Reader threads must be solved: the value might be modified between one read operation and the subsequent one. By introducing the local variable localSharedValue, it is possible to confine the sharedValue to the stack of each Reader thread. Only one read to the shared state is performed during the assignment of localSharedValue, while the subsequent three read operations are performed on the local variable, which cannot be modified by other threads.

Compared to the original version, the program does not require any form of lock, thanks to the properties of volatile variables and the use of the stack confinement technique, allowing reader threads to access the shared state only once in read mode.

```java
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 (A5Exercise1.isRunning.get()) {
      // Read shared value to local variable
      final int localSharedValue = A5Exercise1.sharedValue;
      // Use local value for updates and compare
      if (localValue != localSharedValue) {
        localValue = localSharedValue;
      } else
        System.out.println("Reader" + id + ": (" + localValue + " == " + localSharedValue + ")");
    }
  }
}

public class A5Exercise1 {
  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
      A5Exercise1.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.");
  }
}
```

Solution exercise 2

The program suffers from the "escape of *this* reference" problem when constructing the EventListener class. Because of the "escape of *this* reference", objects of the type EventListener get published while the constructor is still running. This situation can be problematic if the EventSource thread manages to access the fields of the objects while they are not yet fully initialized.

To correct this error you need to refactor the classes, by removing the EventSource type parameter from the constructor of the EventListener class. The registration phase of the listener must be postponed, in order that it happens after the creation of the objects. By doing so, the escape of the "this" reference cannot happen anymore and the objects are safely constructed.

The code should therefore be modified as follows:

```
class EventListener {
  private final int id;

  public EventListener(final int id) {
    // Sleep added to facilitate the appearance of the problem. In a real world
    // program, other initialization operations may be performed
    try {
      Thread.sleep(4);
    } catch (final InterruptedException e) {
      // Thread interrupted
    }

    this.id = id;
  }

  public void onEvent(final int listenerID, final Event e) {
      // Check that the listener's ID called from the eventSource matches the listener's instance's
is
    if (listenerID != id)
      System.out.println("Inconsistent listener ID" + listenerID + " : " + e);
  }
}

class EventListener {
  protected final int id;

  public EventListener(final int id) {

    // Sleep added to facilitate the appearance of the problem. In a real world
    // program, other initialization operations may be performed
    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 A5Exercise2 {
  public static void main(final String[] args) {
    final EventSource eventSource = new EventSource();
    final Thread eventSourceThread = new Thread(eventSource);

    // Start eventSource thread
    eventSourceThread.start();
    // Create and register listeners to eventSource
    final List<EventListener> allListeners = new ArrayList<>();
    for (int i = 1; i <= 20; i++) {
      final EventListener listener = new EventListener(i);
      eventSource.registerListener(i, listener);
      allListeners.add(listener);
    }

    // Wait for thread to terminate
    try {
      eventSourceThread.join();
    } catch (final InterruptedException e) {
      // Thread interrupted
    }
  }
}
```

Solution exercise 3

By comparing the implementations of A5Exercise3A and A5Exercise3B, the only real difference is that the first class uses a non-thread-safe version of the shared state (SharedStateClass), whereas the second class uses a thread-safe variant (ThreadSafeSharedState).

The program's logic is the same, therefore both suffer from a safe-publication problem. For both types of shared states (non-thread-safe and thread-safe) safe-publication must be performed. The reference to the object and the state of the object must be made visible at the same time. During the publication of the object, in the provided programs, no synchronization tool such as synchronized blocks or ReentrantLocks is used. As a consequence the reading thread cannot see the reference to the object correctly and waits indefinitely in the while loop. The thread which builds the shared state publishes the reference, but the reference value is not correctly propagated in the caches of the used processor cores. To correct the problem, the reference must be shared using safe-publication.

Solution using volatile reference for ThreadSafeSharedState

By specifying the A5Exercise5B.sharedState reference as volatile, correct visibility of the reference to all threads is guaranteed.

```
public class A5Exercise3B {
 …
    static volatile ThreadSafeSharedState sharedState = null;
```

Solution using synchronized methods to access ThreadSafeSharedState

The same problem can also be solved by introducing e.g. static synchronized methods (*getSharedState()* and *setSharedState()*) to manage the access to the shared state reference. All direct accesses to the reference need to be replaced by method calls. The *Starter* thread publishes the object using the *setSharedState()* method, while the *Helper* class will access the shared state by using the *getSharedState()* method. The same can be also achieved by using explicit locks instead of intrinsic locks to protect the access to the reference in the *getSharedState()* and *setSharedState()* methods.

Important

When using non-thread-safe objects (*SharedStateClass*), it is also required to protect concurrent accesses to the shared state itself. The solution with volatile reference only guarantees correct visibility of the reference but does not influence the thread-safety of the referenced object. The same applies to the previous solution with synchronized methods. The *getSharedState* and *setSharedState* methods only control/protect access to the reference and not to the state of the referenced object.

Solution using ReadWriteLocks for SharedStateClass (non-thread-safe shared state)

As stated in the previous "Important" paragraph both the reference to *sharedState* and the shared state's contents must to be protected. The proposed solution uses a ReadWriteLocks to achieve this. ReadLocks are used to check when the reference is set and to retrieve the shared state's value (using the *getValue* method). WriteLocks are instead used when setting the reference (publishing the object) and when performing changes to the shared state (using the *increment* method).

```
private static final ReadWriteLock lock = new ReentrantReadWriteLock();
private static SharedStateClass sharedState = null;
```

```
static class Helper implements Runnable {
    @Override
    public void run() {
        System.out.println("Helper : started and waiting until shared state is set!");
        boolean sharedStateIsSet = false;
        while (!sharedStateIsSet) {
            lock.readLock().lock();
            try {
                 sharedStateIsSet = A5Exercise3A.sharedState != null;
            } finally {
                lock.readLock().unlock();
            }
        }

        int lastValue;
        lock.readLock().lock();
        try {
            lastValue = A5Exercise3A.sharedState.getValue();
        } finally {
            lock.readLock().unlock();
        }

        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;
            lock.readLock().lock();
            try {
                curValue = A5Exercise3A.sharedState.getValue();
            } finally {
                lock.readLock().unlock();
            }

            if (lastValue != curValue) {
                lastValue = curValue;
                break;
            }
        }
        System.out.println("Helper : value changed to " + lastValue + "!");

        for (int i = 0; i < 5000; i++) {
            lock.writeLock().lock();
            try {
                A5Exercise3A.sharedState.increment(ThreadLocalRandom.current().nextInt(1, 10));
            } finally {
                lock.writeLock().unlock();
            }

            if ((i % 100) == 0)
                try {
                    Thread.sleep(1);
                } catch (final InterruptedException e) {
                }
        }
        System.out.println("Helper : completed");
    }
}

static class Starter implements Runnable {

    @Override
    public void run() {
        System.out.println("Starter: sleeping");
        try {
            Thread.sleep(1000);
        } catch (final InterruptedException e) {
        }

        System.out.println("Starter: initialized shared state");

        lock.writeLock().lock();
        try {
            A5Exercise3A.sharedState = new SharedStateClass();
        } finally {
            lock.writeLock().unlock();
        }


        // Sleep before updating
        try {
            Thread.sleep(1000);
        } catch (final InterruptedException e) {
        }

        // Perform 5000 increments and exit
        System.out.println("Starter: begin incrementing");
        for (int i = 0; i < 5000; i++) {

            lock.writeLock().lock();
            try {
                A5Exercise3A.sharedState.increment(ThreadLocalRandom.current().nextInt(1, 10));
            } finally {
                lock.writeLock().unlock();
            }

            if ((i % 100) == 0)
                try {
                    Thread.sleep(1);
                } catch (final InterruptedException e) {
                }
        }
        System.out.println("Starter: completed");
    }
}
```

Solution with “Holder of immutable objects”

To implement this version, it is required to introduce an immutable class (*Holder*) that contains the variables (*count* and *value*) and an ImmutableSharedState class that implements the same methods (*getValue*, *getCount*, *getAverage*, and *increment*) as the other variants of SharedState classes. The proposed solution uses a volatile reference to perform the safe publication. For a more robust solution, an AtomicReference could be used to store the reference to the Holder type object and exploit the CAS idiom in the increment method.

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

    Holder(int count, int value) {
        this.count = count;
        this.value = value;
    }

    public Holder increment(int delta) {
        return new Holder(this.count + 1, this.value + delta);
    }

    public int getValue() {
        return value;
    }

    public int getCount() {
        return count;
    }

    public float getAverage() {
        return getCount() == 0 ? 0 : (float) getValue() / (float) getCount();
    }
}

class ImmutableSharedState {
    private volatile Holder sharedState = new Holder(0, 0);

    public ImmutableSharedState() {
        System.out.println("ImmutableSharedState");
    }
    public void increment(int delta) {
        sharedState = sharedState.increment(delta);
    }

    public int getValue() {
        return sharedState.getValue();
    }

    public int getCount() {
        return sharedState.getCount();
    }

    public float getAverage() {
        return sharedState.getAverage();
    }
}
```