Solution exercise 1

The program is affected by a check-then-act race condition: a decision is taken according to the value of shared unprotected state (*occupied, available*) that can be modified concurrently between the "if" of the decision and the following operation. The correct execution of the operation may therefore be compromised.

The increments of the *totUtilized* variable as well as the increments and decrements of the *occupied* variable are also part of the shared state and are affected by a read-modify-write race condition. The source code of these increments should therefore be protected using the same lock.

Finally, it must be assured that the readings of the *totUtilized* and *totOccupied* variables through the respective public methods are also protected by locks.

Solution with synchronized block

The problem can be solved by using an intrinsic lock (*synchronized* keyword) using the object instance as a lock to protect access to shared variables (since the shared state is composed of an instance variable). As a result, the compound action that checks the status and performs the following operations is atomic.

```
public boolean enter() {

    synchronized (this){

        // Any restrooms available?
        if (numOccupied < numAvailable) {
            // Available: occupy
            numOccupied++;
            usageCount++;
        } else {
            // All occupied!
            occupiedCount++;
            return false;
        }
    }

    // use restroom: no protection required
    useRestroom();

    // leave restroom
    synchronized (this) {
        numOccupied--;
    }
    return true;
}
```

Solution with explicit locks

To solve the exercise with explicit locks, it is first required to instantiate a new ReentrantLock. Then the compound actions need to be protected, by calling the lock / unlock methods. The portions of code to be protected need to be enclosed in a try / finally block.

```
private final Lock lock = new ReentrantLock();

...

public boolean enter() {
    lock.lock();
    try {
        // Any restrooms available?
        if (numOccupied < numAvailable) {
            // Available: occupy
            numOccupied++;
            usageCount++;
        } else {
            // All occupied!
            occupiedCount++;
            return false;
        }
    } finally {
        lock.unlock();
    }

    // use restroom
    useRestroom();

    lock.lock();
    try {
        numOccupied--;
    } finally {
        lock.unlock();
    }
    return true;
}
```

Solution with synchronized methods

To solve the problem with the synchronized methods the *enter* method must be refactored. Each synchronized block becomes a separate synchronized method: the compound action is extracted to the *tryOccupy* method, while the logic to free the bathroom is moved to the *leave* method. As a result, the *enter* method becomes more readable.

```
public boolean enter() {
    if (tryOccupy() == false) {
        return false;
    }

    // use restroom: no protection required
    useRestroom();

    leave();
    return true;
}

private synchronized boolean tryOccupy() {
    // Any restrooms available?
    if (numOccupied < numAvailable) {
        // Available: occupy
        numOccupied++;
        usageCount++;
    } else {
        // All occupied!
        occupiedCount++;
        return false;
    }
    return true;
}

private synchronized void leave() {
    numOccupied--;
}
```

Solution with explicit locks and methods

As an alternative to synchronized methods, the same methods can be protected using explicit locks in the following way.

```
private final Lock lock = new ReentrantLock();

...

public boolean enter() {
    if (tryOccupy() == false) {
        return false;
    }

    // use restroom
    useRestroom();

    leave();
    return true;
}

private boolean tryOccupy() {
    lock.lock();
    try {
        // Any restrooms available?
        if (numOccupied < numAvailable) {
            // Available: occupy
            numOccupied++;
            usageCount++;
            return true;
        } else {
            // All occupied!
            occupiedCount++;
            return false;
        }
    } finally {
        lock.unlock();
    }
}

private void leave() {
    lock.lock();
    try {
        numOccupied--;
    } finally {
        lock.unlock();
    }
}
```

Solution assignment 2

Solution with explicit locks

```
class Sensor implements Runnable {
    private final int threshold;

    public Sensor(final int threshold) {
        this.threshold = threshold;
    }

    @Override
    public void run() {
        System.out.println("Sensor[" + threshold + "]: start monitoring!");

        while (!A2Exercise2.resetIfAbove(threshold)) {
            /* Busy wait */
        }

        System.out.println("Sensor[" + threshold + "]: threshold passed!");
    }
}

public class A2Exercise2 {
    private static final Lock lock = new ReentrantLock();
    private static int amount = 0;

    static int incrementAndGet(final int step) {
        lock.lock();
        try {
            amount += step;
            return amount;
        } finally {
            lock.unlock();
        }
    }

    static boolean resetIfAbove(final int threshold) {
        lock.lock();
        try {
            if (amount > threshold) {
                amount = 0;
                return true;
            }
            return false;
        } finally {
            lock.unlock();
        }
    }

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

        for (int i = 1; i <= 10; i++) {
            final int sensorThreshold = (i * 10);
            threads.add(new Thread(new Sensor(sensorThreshold)));
        }

        // start all threads
        threads.forEach(Thread::start);

        while (true) {
            final int increment = ThreadLocalRandom.current().nextInt(1, 9);
            final int newAmount = incrementAndGet(increment);
            System.out.println("Actuator: shared state incremented by " + newAmount);
            if (newAmount > 120)
                break;
            try {
                Thread.sleep(ThreadLocalRandom.current().nextLong(5, 11));
            } catch (final InterruptedException e) {
                e.printStackTrace();
            }
        }
```

```
        for (final Thread t: threads) {
            try {
                t.join();
            } catch (final InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Solution with Reentrant ReadWriteLock

To implement a solution that uses ReadWriteLocks, the *resetIfAbove* method from the previous solution is split into two methods: the *isAbove* method, which uses the readlock to perform the comparison between the current value and the threshold, and the reset logic, which uses the writelock to perform the reset, in case the *isAbove* method returns true.

This solution might seem more efficient, because the *isAbove* check can be performed in parallel by all Sensors (it uses the read lock). Only when a reset is performed the write lock will be used, blocking any other threads that are trying to acquire the read and the write lock. Unfortunately, this is a common pitfall as it introduces a chech-then-act race condition! Between the invocation of the *isAbove* method and the acquisition of the write lock, the thread may get suspended and in the meantime the value of amount can be change. Therefore, the ReadWriteLock cannot be used to solve this problem.

```
class Sensor implements Runnable {
    private final int threshold;

    public Sensor(final int threshold) {
        this.threshold = threshold;
    }

    @Override
    public void run() {
        System.out.println("Sensor[" + threshold + "]: start monitoring!");

        while (!A2Exercise2.resetIfAbove(threshold)) {
            /* Busy wait */
        }

        System.out.println("Sensor[" + threshold + "]: threshold passed!");
    }
}

public class A2Exercise2 {
    private static int amount = 0;
    private static ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
    private static WriteLock writeLock = lock.writeLock();
    private static ReadLock readLock = lock.readLock();

    static int incrementAndGet(final int step) {
        writeLock.lock();
        try {
            amount += step;
            return amount;
        } finally {
            writeLock.unlock();
        }
    }

    private static boolean isAbove(final int threshold) {
        readLock.lock();
        try {
            return amount > threshold;
        } finally {
            readLock.unlock();
        }
```

```
    }

    static boolean resetIfAbove(final int threshold) {
        // /!\ Check Then Act!
        if (isAbove(threshold)) {
            writeLock.lock();
            amount = 0;
            writeLock.unlock();
            return true;
        }
        return false;
    }

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

        for (int i = 1; i <= 10; i++) {
            final int sensorThreshold = (i * 10);
            threads.add(new Thread(new Sensor(sensorThreshold)));
        }

        // start all threads
        threads.forEach(Thread::start);

        while (true) {
            final int increment = ThreadLocalRandom.current().nextInt(1, 9);
            final int newAmount = incrementAndGet(increment);
            System.out.println("Actuator: shared state incremented by " + newAmount);
            if (newAmount > 120)
                break;
            try {
                Thread.sleep(ThreadLocalRandom.current().nextLong(5, 11));
            } catch (final InterruptedException e) {
                e.printStackTrace();
            }
        }

        for (final Thread t: threads) {
            try {
                t.join();
            } catch (final InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```