Solution exercise 1

Refactor to independent class

First, a class must be created, e.g. A1Exercise1Thread, that extends the parent class Thread and implements the run method of the original version. To keep track of the index of each Thread, the index is provided to the constructor and stored as internal state of the class. In the main method (which is executed by the main thread) the creation of the anonymous inner class is replaced with the instantiation of the new A1Exercise1Thread class. In order to start the newly created threads it is required to invoke the start method for each instance of A1Exercise1Thread. At this point the anonymous inner class has been refactored into an independent class.

```java
class A1Exercise1Thread extends Thread {
    final private int index;

    A1Exercise1Thread(int index) {
        this.index = index;
    }

    @Override
    public void run() {
        long fibo1 = 1, fibo2 = 1, fibonacci = 1;
        for (int i = 3; i <= 700; i++) {
            fibonacci = fibo1 + fibo2;
            fibo1 = fibo2;
            fibo2 = fibonacci;
        }
        System.out.println("Thread: " + index +": " + fibonacci);
    }
}

public class A1Exercise1 {
    public static void main(String[] args) {
        /* Creating threads */
        Collection<Thread> allThreads = new ArrayList<>();
        for (int i = 1; i <= 5; i++) {
            Thread t = new A1Exercise1Thread(i);
            allThreads.add(t);
        }

        /* Starting threads */
        for (Thread t : allThreads)
            t.start();

        /* Waiting for threads to terminate */
        for (Thread t : allThreads) {
            try {
                t.join();
                System.out.println(t + " has terminated");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Refactor to Runnable

To transform the class into a Runnable, instead of extending the Thread class, the Runnable interface needs to be implemented. In the main method, instead of creating directly the threads, new runnables have first to be instantiated. Each runnable is then assigned to a thread (at the time of creation), which will be responsible for execution.

```java
class A1Exercise1Runner implements Runnable {
    final private int index;

    A1Exercise1Runner(int index) {
        this.index = index;
    }

    @Override
    public void run() {
        long fibo1 = 1, fibo2 = 1, fibonacci = 1;
        for (int i = 3; i <= 700; i++) {
            fibonacci = fibo1 + fibo2;
            fibo1 = fibo2;
            fibo2 = fibonacci;
        }
        System.out.println("Runner: " + index +": " + fibonacci);
    }
}

public class A1Exercise1 {
    public static void main(String[] args) {
        /* Creating threads */
        Collection<Thread> allThreads = new ArrayList<>();
        for (int i = 1; i <= 5; i++) {
            final int index = i;
            System.out.println("Main: creating thread " + index);
            A1Exercise1Runner runner = new A1Exercise1Runner(i);
            Thread t = new Thread(runner);
            allThreads.add(t);
        }

        /* Starting threads */
        for (Thread t : allThreads)
            t.start();

        /* Waiting for threads to terminate */
        for (Thread t : allThreads) {
            try {
                t.join();
                System.out.println(t + " has terminated");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Lambda expression

To introduce the lambda expression, the Functional Interface *Runnable* is used, which imposes the run method. When a new Thread is created, instead of instantiating a new class A1Exercise1Runnable, the implementation of the run method is specified directly via a lambda expression.

```java
public class A1Exercise1 {
    public static void main(String[] args) {
        /* Creating threads */
        Collection<Thread> allThreads = new ArrayList<>();
        for (int ii = 1; ii <= 5; ii++) {
            final int index = ii;
            System.out.println("Main: creating thread " + index);

             // Creating a new thread that executes the Runnable created using lambda expression
            final Thread t = new Thread(() -> {
                long fibo1 = 1, fibo2 = 1, fibonacci = 1;
                for (int i = 3; i <= 700; i++) {
                    fibonacci = fibo1 + fibo2;
                    fibo1 = fibo2;
                    fibo2 = fibonacci;
                }
                System.out.println("Lambda" + index + ": " + fibonacci);
            });

            allThreads.add(t);
        }

        /* Starting threads */
        for (Thread t: allThreads)
            t.start();

        /* Waiting for threads to terminate */
        for (Thread t: allThreads) {
            try {
                t.join();
                System.out.println(t + " has terminated");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Solution assignment 2

The program suffers from a read-modify-write race condition that occurs while incrementing the *entrances*, *exits* and *tolls* variables of the shared state *highway*. Moreover, the *exits* and *tolls* variables are interdependent, which means that the two increments form a compound action that needs to be performed in an atomic way.

Solution with synchronized block

In this solution a synchronized block must be introduced around the portion of code that must be atomic, using the shared state instance variable *highway* as synchronization object. The operation of *driveHighwaySection* between the increment operations does not need protection, since it does not access the shared state.

```
class Driver implements Runnable {
...
    @Override
    public void run() {
        System.out.println("Driver " + id + ": started");

        for (int i = 0; i < 500; i++) {
            enterHighway();

            synchronized (highway) {
                highway.entrances++;
            }
            highway.entrances++;

            int sectionToll = driveHighwaySection();

            synchronized (highway) {
                highway.exits++;
                highway.tolls += sectionToll;
            }
            tollsPaid += sectionToll;
        }
        System.out.println("Driver " + id + ": finished");
    }
```

Solution with synchronized methods

To solve the problem with the synchronized methods, the logic of the two synchronized blocks is extracted into two new methods of the *Highway* class: *enter* and *exit*. The methods must be declared as synchronized in order to use the instance of the class as the synchronization object, in a similar ways as it has been done for the previous solution. Then, the logic within the run method of the *Driver* class can be simplified, replacing it with the invocation of the two methods. Finally, the shared *entrances*, *exits* and *toll* variables can be made private, because all accesses are performed exclusively through the *enter* and *exit* instance methods.

```
class Driver implements Runnable {

...
    @Override
    public void run() {
        System.out.println("Driver " + id + ": started");

        for (int i = 0; i < 500; i++) {
            enterHighway();

            highway.enter();

            final int sectionToll = driveHighwaySection();

            highway.exit(sectionToll);
            tollsPaid += sectionToll;
        }
        System.out.println("Driver " + id + ": finished");
    }

class Highway {
    private int entrances = 0;
    private int exits = 0;
    private int tolls = 0;

    public synchronized void enter() {
        entrances++;
    }

    public synchronized void exit(final int toll) {
        this.tolls += toll;
        this.exits++;
    }

    public synchronized int getEntrances() {
        return entrances;
    }

    public synchronized int getExits() {
        return exits;
    }

    public synchronized int getTolls() {
        return tolls;
    }
}
```

Solution with explicit locks

For this solution a ReentrantLock is introduced, which for convenience can be declared directly in the *Highway* class, in order to explicitly link it to the shared state. Then, as for the solution with synchronized blocks, the compound actions must be protected, enclosing them between the lock() and unlock() method calls. In addition, it is good practice to enclose the portion of code to be protected in a try / finally block to ensure that the lock is always released, especially in case of thrown exceptions inside the locked portion of code.

```
class Highway {
    public final Lock lock = new ReentrantLock();
    public int entrances = 0;
    public int exits = 0;
    public int tolls = 0;
}

class Driver implements Runnable {

    @Override
    public void run() {
        System.out.println("Driver " + id + ": started");

        for (int i = 0; i < 500; i++) {
            enterHighway();
            highway.lock.lock();
            try {
                highway.entrances++;
            } finally {
                highway.lock.unlock();
            }
            int sectionToll = driveHighwaySection();

            highway.lock.lock();
            try {
                highway.exits++;
                highway.tolls += sectionToll;
            } finally {
                highway.lock.unlock();
            }
            tollsPaid += sectionToll;
        }
        System.out.println("Driver " + id + ": finished");
    }
```

Solution with methods and explicit locks

This solution is very similar to the one with synchronized methods. Instead of declaring the methods as synchronized, a ReentrantLock instance is added to the *Highway* class and used as lock to protect the compound actions in the *enter* and *exit* methods. In the *Driver's* run method on the other hand, the code remains exactly the same as in the version with synchronized methods. The code accessing the shared state is completely encapsulated and protected within the *Highway* class.

```
class Highway {
    private final Lock lock = new ReentrantLock();
    private int entrances = 0;
    private int exits = 0;
    private int tolls = 0;

    public void enter() {
        lock.lock();
        try {
            entrances++;
        } finally {
            lock.unlock();
        }
    }

    public void exit(final int toll) {
        lock.lock();
        try {
            this.tolls += toll;
            this.exits++;
        } finally {
            lock.unlock();
        }
    }

    public int getEntrances() {
        lock.lock();
        try {
            return entrances;
        } finally {
            lock.unlock();
        }
    }

    public int getExits() {
        lock.lock();
        try {
            return exits;
        } finally {
            lock.unlock();
        }
    }

    public int getTolls() {
        lock.lock();
        try {
            return tolls;
        } finally {
            lock.unlock();
        }
    }
}
```

```java
class Driver implements Runnable {

    @Override
    public void run() {
        System.out.println("Driver " + id + ": started");

        for (int i = 0; i < 500; i++) {
            enterHighway();

            highway.enter();

            final int sectionToll = driveHighwaySection();

            highway.exit(sectionToll);
            tollsPaid += sectionToll;
        }
        System.out.println("Driver " + id + ": finished");
    }
```

Solution exercise 3

```java
class Account {
    public final Lock lock;
    public long accountBalance;

    public Account(final int initialAmount) {
        accountBalance = initialAmount;
        lock = new ReentrantLock();
    }
}

class User implements Runnable {
    private final int ID;
    private final int delay;
    private long wallet;
    private final Random random;

    private final Account account;

    public User(final Account bankAccount, final int id, final int delay) {
        this.account = bankAccount;
        this.ID = id;
        this.delay = delay;
        this.wallet = 0;
        this.random = new Random();
        log("created. Withdrawing every " + delay + " ms");
    }

    @Override
    public void run() {
        boolean isRunning = true;

        while (isRunning) {
            try {
                // Wait before continuing execution
                Thread.sleep(delay);
            } catch (final InterruptedException ex) {
                return;
            }

            // Compute the amount to withdraw [5, 50]
            final long requestedAmount = random.nextInt(46) + 5;
            final long withdrawnAmount;
            final long startBalance;

            // Request exclusive access to shared account
            account.lock.lock();
            try {
                startBalance = account.accountBalance;

                if (account.accountBalance == 0) {
                    withdrawnAmount = 0;
                } else if (account.accountBalance < requestedAmount) {
                    withdrawnAmount = account.accountBalance;

                    // Take all remaining
                    account.accountBalance = 0;
                } else {
                    withdrawnAmount = requestedAmount;

                    // Update balance
                    account.accountBalance -= requestedAmount;
                }
            } finally {
                // Release lock
                account.lock.unlock();
            }

            // Update user's wallet with withdrawn money
            wallet += withdrawnAmount;

            // Print to console, no need to keep holding the lock
            if (withdrawnAmount == 0) {
                log("bank account empty!");

                // Terminate execution
                isRunning = false;
            } else if (withdrawnAmount < requestedAmount) {
                log("could withdraw only " + withdrawnAmount + "$ from bank account instead of "
                                           + requestedAmount + "$");

                // Terminate execution
                isRunning = false;
            } else
                // Notify successful withdrawal
                log("withdrawing " + requestedAmount + "$ from bank account having "
 + startBalance + "$. New account balance: " + (startBalance - requestedAmount) + "$");
        }
    }

    public long getWallet() {
        return wallet;
    }

    public void log(final String message) {
        System.out.println("User " + ID + ": " + message);
    }
}

public class A1Exercise3 {
    static final int INITIAL_AMOUNT = 10000;
    static final int USERS = 5;

    public static void main(final String[] x) {
        final Random random = new Random();
        final Account account = new Account(INITIAL_AMOUNT);

        final ArrayList<User> users = new ArrayList<User>();
        final ArrayList<Thread> allUserThread = new ArrayList<Thread>();
        for (int i = 0; i < USERS; i++) {
            // Generate random number bewteen 5 and 20
            final int delay = 5 + random.nextInt(16);
            final User curUser = new User(account, i, delay);
            users.add(curUser);
            allUserThread.add(new Thread(curUser));
        }
        System.out.println("--------------------");

        try {
            // Start all threads
            for (final Thread t: allUserThread)
                t.start();

            // Wait until all threads have completed
            for (final Thread t: allUserThread)
                t.join();
        } catch (final InterruptedException e) {
            // No exception handling
        }
        System.out.println("--------------------");
        long totalUserCash = 0;
        for (final User u : users) {
            final long userWallet = u.getWallet();
            totalUserCash += userWallet;
            u.log("has withdrawn : " + userWallet + "$");
        }
        System.out.println("Total withdrawn by users : " + totalUserCash + "$");
        final long balance = account.accountBalance;
        System.out.println("Final account balance :" + balance + "$");
        System.out.println("Sum of total withdrawn and final account balance: "
                + (totalUserCash + balance) + "$");
        System.out.println("Initial bank account balance  : " + INITIAL_AMOUNT + "$");

        System.out.println("--------------------");
        System.out.println("Simulation finished.");
    }
}
```