Bidirectional Transformations - Exercises

As there is no specific requirements regarding implementation, I will be using pseudocode.

Exercise 1
a)

We can get the pair of integers and produce their product:

```
get(n,m) = n*m
```

We can write a put that updates one or both of the numbers in the int pair, that will have the given product:

```
put((n,m), v') =
     if (v' / n) mod 1 == 0 then (n,(v'/n))
     else if (v' / m) mod 1 == 0 then ((v'/m),m)
     else (v',1)
```

b)

Here we can write a get in both directions, as both sides can be generated from the other.
S: Person entity
V: Person
get: S -> V:

```
get(s) = new Person(s.firstName + " " + s.lastName)
```

S: Person
V: Person entity
get: S -> V:

```
get(s) =
     new PersonEntity(s.split(" ")[0], s.split(" ")[1])
```

c)

Given a list of items (a, b, b, c) you can't generate a single set of unordered, unique items. This will result in the loss of data as one b would have to be removed. Therefore, I assume that there is no requirement of a 1-to-1 transformation where a list must be transformed into a single set, but can rather be multiple sets.

With this assumption, we can define a get where we get sets, and generate a list as such:

```
get(sets) = {
     new List list;
     sets.forEach(set -> list.addAllItemsFromSet(set));
     list.sort();
}
```

We can also define a get where we generate unique sets from a list. As there are multiple sets, I return a list of sets:

```
get(list) = {
     new List listOfSets
     new Set set;
     new List listOfDuplicates;
     new List recursiveList;
     for (item i : list) {
          if (!set.contains(i)) {
               set.add(i);
          } else {
               listOfDuplicates.add(i);
          }
     }
     recursiveList = get(listOfDuplicates);
     listOfSets.add(set);
     for (Set s : recursiveList) {
          listOfSets.add(s);
     }
}
```

Exercise 2

I propose an asymmetric lens. You can generate the datamodel based on the classmodel, but not vice versa. This is because operations are not mapped to the datamodel.

get(classmodel) -> datamodel
put(classmodel, datamodel) -> classmodel

The implementation of get will simply use the mappings defined in the task.

The task states that associations with target PrimType are stored as columns. As an association has a source as well as a target, I assume this means that the source is stored along with the PrimType even though this is not shown in Figure 1. With this assumption, we have all the information needed to reverse the OR-mapping and implement put as this reversed ORM.

Exercise 3

a)

Correctness:
Given an A1' and an A2 which are not consistent, the forward restoration R-> must restore consistency.
I assume that A1 and A2 are in a consistent state. So when we introduce a change to A1 -> A1', a restoration to a consistent state involves adding the changes from A1' to A2.
Say you have a composer ("Bach", 1658, "German") in A1' and there is no composer in A2 with name="Bach" and nationality="German". Then R-> adds Bach to A2. At this stage, all composers in A1' are also in A2. Now say that there is a composer in A2 which is not in A1'. In this case, the changes from the first step have already been added to A2. There was also no changes to A2 before the restoration. So now, all composers in A2, which are not reflected with the same name and nationality in A1' are deleted from A2. A1' and A2 are consistent => correctness justified.

The same steps can be applied to the backwards restoration <-R. This implies correctness for both restorations.

Hippocraticness:
Given two models A1 and A2 which are already consistent, R-> and <-R should do nothing.
As every step in the restorations which perform any additions or deletions to a model are dependent on there being inconsistencies between the models - no changes are made when the models are consistent => hippocraticness justified.

b)

R->(A1, R->(A1', A2)) = A2.
R->(A1', A2) = A2'. So the above equation can be rewritten to R->(A1, A2') = A2.
This describes changing something back to a previous state of consistency. Say we have two models A1 and A2 which are consistent. If we modify A1 -> A1' and perform the restoration R->(A1', A2), we modify A2 to a new consistent state A2'. If we preserved the initial state of A1 (before we modified it to A1') and perform a new restoration on the now modified A2' - we will essentially revert A2 back to the state where it was consistent with the initial A1.

This is not satisfied in the composer example. In the restoration in the composer example, a composer may be deleted from A2 in order to produce A2'. Say the composer ("Joseph", "Haydn", "German") is deleted from A2 as part of a restoration. If you try to re-introduce "Haydn" as part of a restoration from an old state, it must be added to R2' as a new composer based on the data ("Haydn", 1732, "German"). This will add the composer ("", "Haydn", "German") to R2'. Data has been deleted which cannot be re-added. R2 =/ R2'.

Exercise 4

a)

I assume this question asks why the lens is asymmetric. This is because you cannot generate two unique sets X and Y with consistency based on V and get(X,Y), as get(X,Y) only states which sets are shared between X and Y, but contains no information about where the not-shared-data should go.

b & c)

I don't know how I am supposed to solve these tasks

Exercise 5

I'm afraid I don't understand the questions here.. I hope someone will publish a solution to this oblig.