Object-Oriented Programming 2

Rolf Haenni & Andres Scheidegger

Exercises 2

1. Series of Values

Consider the following two generic functional interfaces `Function` and `Predicate`:

```
1 public interface Function<V> {
2   public V apply(V value);
3 }
4
5 public interface Predicate<V> {
6   public boolean test(V value);
7 }
```

Write a generic class `Series<V>`, which allows to generate finite series of values for different types. The class must provide the following methods:

- `boolean hasNext()`: checks if the series contains at least one additional value
- `V next()`: returns the next value in the series (if one exists)
- `void reset()`: resets the series to its initial state
- `List<V> toList()`: returns a list that contains all the values of the series

A series can be specified in two different ways:

1. Initial value, update function, condition

   Example: 0, $f(x) = x + 1$, $x \leq 10$ for $\langle 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 \rangle$

2. Initial value, update function, final value

   Example: 0, $f(x) = x + 2$, 10 for $\langle 0, 2, 4, 6, 8, 10 \rangle$

Define corresponding constructors to support both types of series (think of one of them as a special case of the other).

Here is a test program:

```
1  public class SeriesTester {
2
3    public static void main(String[] args) {
4
5      Series<Integer> s1 = new Series<>(0, x -> x + 1, 10);
6      Series<Integer> s2 = new Series<>(0, x -> x + 2, x -> x <= 10);
7      Series<Integer> s3 = new Series<>(1, x -> x + x, x -> x <= 80);
8      Series<String> s4 = new Series<>("", s -> s + "*", s -> s.
           length() <= 10);
9      Series<String> s5 = new Series<>("HelloWorld", s -> s.substring
           (0, s.length() - 1), s -> !s.equals(""));
10
11     int i = 1;
12     for (Series<?> series : new Series[] { s1, s2, s3, s4, s5 }) {
13       System.out.print("s" + (i++) + " = ");
14       System.out.println(series.toList());
15     }
16   }
17 }
```

This is the output generated by the test program:

```
0 1 2 3 4 5 6 7 8 9 10
s1 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
s2 = [0, 2, 4, 6, 8, 10]
s3 = [1, 2, 4, 8, 16, 32, 64]
s4 = [, *, **, ***, ****, *****, ******, *******, ********, *********,
    **********]
s5 = [HelloWorld, HelloWorl, HelloWor, HelloWo, HelloW, Hello, Hell,
   Hel, He, H]
```

2. Series of Pairs

Take the generic class `Pair<S,T>` from the course slides. Write the following classes:

- `PairFunction<S,T>`, which implements the `Function<V>` interface, but works for pairs of type `Pair<S,T>` instead of single values of type `V`. Write a constructor, which expects two arguments `f1` of type `Function<S>` and `f2` of type `Function<T>` as input. When the function is applied to a pair, `f1` is applied to the first value of the pair and `f2` to the second of the pair. The two results form the new pair.
- `PairPredicate<S,T>`, which implements the `Predicate<V>` interface, but works for pairs of type `Pair<S,T>` instead of single values of type `V`. Write a constructor, which expects two arguments `p1` of type `Predicate<S>` and `p2` of type `Predicate<T>` as input. The predicate returns true for an input pair, if `p1` returns true for the first value and `p2` returns true for the second value of the pair.

Write a generic class `PairSeries<S,T>`, which inherits from `Series<V>`. Its purpose is to produce a series of pairs of type `Pair<S,T>`. Write a constructor, which takes two series—one of type `S` and one of type `T`—as input and produces a corresponding series of pairs (its length corresponds to the shorter of the two input series).

Here is a test program:

```
1  public class PairSeriesTester {
2
3    public static void main(String[] args) {
4
5      Series<Integer> s1 = new Series<>(0, x -> x + 2, x -> x <= 10);
6      Series<String> s2 = new Series<>("", s -> s + "*", s -> s.
           length() <= 10);
7
8      PairSeries<Integer, String> s12 = new PairSeries<>(s1, s2);
9      System.out.println(s12.toList());
10   }
11 }
```

This is the output generated by the test program:

```
[[0,], [2,*], [4,**], [6,***], [8,****], [10,*****]]
```

3. The Comparable Interface

The generic Java interface `Comparable<T>` is used in methods such as `Collections.sort` or `Arrays.sort` to determine the *natural order* of the elements to be sorted. The classes `String`, `Integer`, `BigInteger`, etc. all implement this interface, for example `String` implements `Comparable<String>`.

a) Define a new class `ComparablePair`, which inherits from `Pair` and implements the `Comparable` interface. Make sure that both generic types of your class also implement the `Comparable` interface. This allows you to define the natural order of a pair based on the natural order of its values by assuming that the order of the first value "dominates" the order of the second value. Examples: $(3,4) < (6,2)$, $(3,4) > (1,2)$, $(3,4) < (3,6)$, $(3,4) > (3,2)$, etc.

   Test your solutions by calling `Collections.sort` for a list of comparable pairs.

b) Write two generic static methods `getMin` and `getMax`, which expect two comparable pairs as inputs and return the minimum respectively maximum of the two (relative to their natural order).

Here is a test program:

```
1    public static void main(String[] args) {
2      ComparablePair<Integer,String> p1 = new ComparablePair<>(3, "
           Hello");
3      ComparablePair<Integer,String> p2 = new ComparablePair<>(3, "
           World");
```

```
4       ComparablePair<Integer,String> p3 = new ComparablePair<>(2, "
            Hello");
5       ComparablePair<Integer,String> p4 = new ComparablePair<>(2, "
            World");
6       ComparablePair<Integer,String> p5 = new ComparablePair<>(4, "
            Hello");
7       ComparablePair<Integer,String> p6 = new ComparablePair<>(4, "
            World");
8
9       System.out.println(ComparablePair.getMin(p1, p2));
10      System.out.println(ComparablePair.getMax(p1, p2));
11
12      List<ComparablePair<Integer,String>> list = new ArrayList<>();
13      list.add(p1);
14      list.add(p2);
15      list.add(p3);
16      list.add(p4);
17      list.add(p5);
18      list.add(p6);
19      Collections.sort(list);
20      System.out.println(list);
21  }
```

This is the output generated by the test program:

```
[3,Hello]
[3,World]
[[2,Hello], [2,World], [3,Hello], [3,World], [4,Hello], [4,World]]
```