Object-Oriented Programming: Interfaces and Inheritance in Java

Assignment III: Coin Set

In this Assignment we shall look at how to implement Interfaces using a simple Coin Set example. We shall also review some rules that govern *Sets* as *Collections*. A `CoinSet` object can contain a set of positive integers, i.e., a collection of coin values (integers) where each integer can occur only once, such a set might be {1, 2, 55, 78, 100, 4} or {1, 2, 5, 7} but not {1, 1, 20}. I have written an interface for this data structure, `simpleCoinSetInt`, with the following methods:

```
boolean isEmpty();

void add(int value);

void remove(int value);

int getCount();

int[] toArray();

String toString();
```

`isEmpty()` returns *true* if the set has no elements; `add()` puts an element into the set (if it's not there already); `remove()` deletes the given element from the set; `getCount()` returns the number of items (coin values) in the set; `toString()` returns the set, represented as a string; and `toArray()` returns the set, represented as an array, e.g., for the Ugandan case, we would have {50, 100, 200, 500, 1000}.

a) Your first task is to implement the Interface and define all of these methods. Your class should be called `simpleCoinSet`.

b) Implement a constructor in your `simpleCoinSet` that takes the capacity of the coin set and creates a `CoinSet` object of that size.

c) I have written a test program to test your `CoinSet`. Use it to check if your object is working properly. The test class, `simpleCoinSetTest`, should work without any modifications. Of course, it will not compile until all the above methods are defined in the class `simpleCoinSet`. On addition, you may have to implement `intersect()` and `union()` or comment out the lines that call these function in simpleCoinSetTest for it to compile without errors.

d) On addition to implementing the interface methods, define and implement the following two methods:
```
simpleCoinSet intersect(simpleCoinSet other);
simpleCoinSet union(simpleCoinSet other);
```
`intersect()` returns the intersection of this set and another whereas `union()` returns the union.

e) The test program checks your object with a possible implementation of a coin set. It uses some algorithm to return the smallest number of coins, given an amount and a coin set. E.g., given 2700 and the Ugandan coin set above, it should return [200, 500, 1000, 1000]. This algorithm, written in *Changer.java*, just works and shouldn't be modified at all.

f) As you may notice, some coin sets may not always be possible to propose the number of coins amounting to the provided value. E.g., with the coin set above, it is impossible to suggest the number of coins that amount to 2756. In order to fix this, and not to return garbage in case of non-corresponding input, make sure that your coin set always contains a value of 1 (one). That way, you'll then be able to return [1, 1, 1, 1, 1, 1, 50, 200, 500, 1000, 1000].

g) Using the provided test program, check your `CoinSet` with the *random*-tester (which randomly generates coin values and amounts to test), *self*-tester (which allows you to input the coin values and amount(s) to test), *intersection*-tester (which tests your intersection implementation), and the *union*-tester (which will check that your union implementation is working fine).

Challenge for the Bored

Modify your implementation above such that it is not necessary to ensure that the coin set has a 1. Instead, with an amount of 2756, you return [1000, 1000, 500, 200, 50] and ignore the unavailable "change". That implies that you write your own `Changer` algorithm and use it to do this job.

This assignment is due during the double lecture for the week of 24th – 28th April, 2017.