CSci 4651 Fall 2008
Problem Set 8: Data Abstraction; Object-Oriented and Generic Programming.
Due Friday, November 14

Problem 1 (8 points). Finish and submit the ML exceptions lab (a part of Monday, November 3rd class).

Problem 2 (32 points, Work in pairs). For this problem you need to write two different implementations of multisets (sets that can have repeated elements). You also need to write an interface for multisets.

Background on multisets. A multiset (also known as a bag) is a set that may contain the same element more than once. For instance, the multiset $M_1 = \{a, b, b\}$ has one copy of a and two copies of b. The order in which elements appear in a multiset does not matter.

The number of times an element appears in a multiset is called its *multiplicity*, denoted by $m()$. For instance, in the multiset M_1 above $m(a) = 1$, $m(b) = 2$. The set of all elements that appear in a multiset is called the *underlying set*. For instance, the underlying set of M_1 is $\{a, b\}$ (this is just a regular set, so each element appears once there).

The class structure. You need to write two generic implementations of multisets: `MultisetElements<T>` and `MultisetMultiplicity<T>`, where `T` is the type of the elements. The classes must implement an interface `Multiset<T>`.

Below is the list of classes and methods that you need to implement:

- `MultisetElements<T>` that uses an array or a list or a vector to store the elements. A duplicated element appears several times on this list.
- `MultisetMultiplicity<T>` stores each element only once and stores multiplicities for each element occurring in the multiset. You may define an inner class to store an element and its multiplicity or you may store multiplicities separately from elements (in a separate array or a list), whatever is more convenient.
- `Multiset<T>` - a Java interface that the client programs will use to refer to multisets. The interface mandates the following methods:
  1. `void add(T c)` - adds c to the multiset.
  2. `int multiplicity(T c)` - returns the multiplicity of c (i.e. the number of times c occurs in the multiset). Returns 0 if c is not in the multiset.
  3. `Multiset<T> union(Multiset<T> ms)` - returns the union of this multiset and `ms`. The union of two multisets M_1 and M_2 is defined as a multiset M_3 with the underlying set $A_1 \cup A_2$ (where A_1 and A_2

are the underlying sets of M_1 and M_2, respectively), and $m_3(a) = max(m_1(a), m_2(a))$. For instance, $\{a, b, b, c\} \cup \{a, c\} = \{a, b, b, c\}$ (note that there is only one a in the result).

The type of the returned multiset is the same as the type of `this` multiset.

The union method should access the other set (the one given as the parameter) through the interface since you don't know what its implementation class is. However, **as an extra credit (up to 10 points)** you can write an implementation that checks if it is of the same class as your class and if it is, access the elements of it directly. for instance, the union method of `MultisetElements` should check if the parameter is an instance of `MultisetElements` (using the `instanceof` operator), and if it is, typecast it to the `MultisetElements` type and access its list of elements directly. To get full credit for this part, implement the **three** methods `union`, `intersection`, and `isSubset` for both Multiset classes in this manner.

4. `Multiset<T> intersect(Multiset<T> ms)` returns the intersection of this multiset and `ms`. The union of two multisets M_1 and M_2 is defined as a multiset M_3 with the underlying set $A_1 \cap A_2$ (where A_1 and A_2 are the underlying sets of M_1 and M_2, respectively), and $m_3(a) = min(m_1(a), m_2(a))$. For instance, $\{a, b, b\} \cap \{a, b, c\} = \{a, b\}$
   The type of the returned multiset is the same as the type of `this` multiset.
5. `boolean isSubset(Multiset<T> ms)` - returns true if this multiset is a subset of `ms`, false otherwise.
6. `T [] getElements()` that returns all elements of the multiset as an array (in any order), each element appearing **only once** (no duplicates).

In addition the two implementation classes may provide a constructor to create an empty multiset.

Usage and testing. Write a main method or a JUnit test class to test that the methods `union`, `intersect`, and `isSubset` work correctly in all 4 possible cases:

1. When called on `MultisetElements` with a `MultisetElements` parameter,
2. When called on `MultisetElements` with a `MultisetMultiplicity` parameter,
3. When called on `MultisetMultiplicity` with a `MultisetElements` parameter, and
4. When called on `MultisetMultiplicity` with a `MultisetMultiplicity` parameter.

Also make sure that you can create multisets of different types (say, Integer and String).

Note that the constructors cannot be called through the interface, so they have to be called in the testing code explicitly, thus breaking encapsulation. There are ways to hide constructors from a client, but they are not perfect and not required here.