Permutation Generator API Design Document

Design Overview

This API aims to provide a simple and easy-to-use permutation generator. The generator is able to generate all possible permutations of a given set of objects of a user specified type. It is also able to accept a user defined predicate so that only permutations satisfying this predicate will be generated. After generating all possible permutations, the user has to reset the generator so that it can be used again. The generator cannot be constructed with an empty list, in which case an IllegalArgumentException will be thrown. This class is made to be thread-safe (at least hopefully so) considering the user might want to process permutations with multiple threads because of the large number of permutations.

The APIs are designed as:

```
class permGenerator {

    /* construct a permGenerator that is going to generate all possible permutations */
    public permGenerator(List);

    /* construct a permGenerator that is going to generate all permutations that satisfy
    Some user defined predicate
    */
    public permGenerator(List, Predicate);

    /* reset the permGenerator with current predicate*/
    public void reset();

    /* reset the permGenerator with new predicate */
    Public void reset(Predicate)

    /* returns a new permutation that is desired */
    public List getNextPerm();
}
```

Use Cases

1. The user wants to store all possible permutations in a matrix form

```
ArrayList<ArrayList<T>> allpermutations;
ArrayList<T> mylist;
permGenerator<T> generator = new permGenerator<>(mylist);

List<T> perm;
while ((perm = generator.getNextPerm()) != null) {
    allpermutations.add(perm);
}
```

2. The user wants to compute information from all permutations that satisfy a predicate

```
ArrayList<T> mylist;
Predicate mypredicate;
permGenerator<T> generator = new permGenerator<>(mylist, mypredicate);

int result = 0;
List<T> perm;
while ((perm = generator.getNextPerm()) != null) {
    result += mylist.get(perm.get(0)).somefiled;
}

return result;
```

3. The user wants to do work with multiple threads

```
ArrayList<T> list;
permGenerator<T> generator = new permGenerator<>(list);
void run() {
    int local_count = 0;
    List<T> perm;
    while ((perm = generator.getNextPerm()) != null) {
        local_count += 1;
    }
    System.out.println(this.id + ":" + local_count);
}
void main(String[] args) {
    for (int i = 0; i < 3; ++i) {
        (new concurrent(i)).start();
    }
}
```

Issues List

1. I decided to use List<T> as input type because it provides indexed access efficiently and provides better type safety than arrays

2. A Predicate is used as predicate input type because it is a natural Java construct.

3. I decided to store a copy of the List passed into the constructor in case the user accidentally modified that List, and also because we ARE going to modify that List.

4. The similar reason in 2, I decided to return a copy of the permutation generated from the main API getNextPerm(). Also, only a copy of underlying List is passed into user defined Predicate for the same reason.

5. Considering the obvious need of processing a large amount of permutations in parallel, I tried to make the API thread-safe. One ReentrantLock is used to ensure that each call to reset() and getNextPerm() is atomic.

6. The generator is designed to not reset itself automatically after looping through all possible permutations once so that the user need not bookkeep this information manually, especially when they use a Predicate and they do not know exactly how many different permutations would be generated.

7. The user is allowed to reset the generator with a different Predicate but is not allowed to change the Predicate without resetting the generator, the main reason being that such behavior is not so well defined.