For the reusable permutation generator, the constructor takes as input an ArrayList of any reference type, while primitive type can be substituted as boxed types. I applied generics programming in Java in order to achieve this functionality. The generator then stores this list and by invoking the *generate* method, all permutations are then generated and stored in another private ArrayList called permutationArray with a standard recursive permutation algorithm. Frankly speaking, I understand that it is better to not store the permutation results in memory since as the order grows, it would take up too much internal memory. I tried to find a way to record the status of the generator and then come back and generate the next permutation, but I was not able to do it in the end.

The generator applies the iterator pattern, which means there's another class called PermutationIterator which implements the Iterator interface and has 3 standard public methods, *hasnext*, *next*, and *remove*. The iterator take as input a permutation instance and in accordance with the generator, the iterator is generic as well, which means it can return an ArrayList of whichever type in the permutation.

As for the cryptarithm solver part, I would first point out that Windows' cmd parse * as all files in the current path, so I have to replace * by '*' in my cmd line input. For example, the test program in the class CryptarithmSolve should be run with such kind of input arguments: NORTH '*' WEST = SOUTH '*' EAST.

For the design part, the core class in my design is CryptarithmExpressionContext. The context class not only stores a mapping from letters to digits, but also has a private permutation generator which enables it to update over all possible mappings through the *updateContext* method, with the help of another field bitVecList for storing all available digit combinations. With a proper context, the Cryptarithm class can therefore split the 2 expressions on each side of an equation and evaluate their value in the *evalExpr* method from left to right. Finally in the test program *CryptarithmSolve.main* the cryptarithm can be solved by iterating through all permutations and find the ones that satisfies the input equation.