```
import java.util.ArrayList;
import java.util.Random;

public class ArrayListExercises {

	public static void main(String[] args) {
	}

	/**
	 * Removes all of the strings of even length from the given list
	 * @param listOfStrings the list of Strings (list can be empty)
	 * @return the given list with all even length strings removed
	 */
	public static ArrayList<String> removeEvenLength(ArrayList<String>
listOfStrings)  {
		for (int i = 0; i < listOfStrings.size(); i++) {
			if (listOfStrings.get(i).length() % 2 == 0) {
				listOfStrings.remove(listOfStrings.get(i));
				i--;
			}
		}
		return listOfStrings;   // This return statement should be last
	}

	/**
	 * Moves the minimum value in the list to the front, otherwise preserving
the order of the elements
	 * @param listOfIntegers the list of Integers (list cannot be empty)
	 * @return the given list with the minimum value in the front (zeroth
element)
	 */
	public static ArrayList<Integer> minimumToFront(ArrayList<Integer>
listOfInts)  {
		int min = Integer.MAX_VALUE;
		int minPos = -1;

		for (int i = 0; i < listOfInts.size(); i++) {
			if (min > listOfInts.get(i)) {
				min = listOfInts.get(i);
				minPos = i;
			}
		}

		listOfInts.remove(minPos);
		listOfInts.add(0, min);
		return listOfInts;   // This return statement should be last
	}

	/**
```

```
       * Removes all elements from the given list whose values are in the range
min through max (inclusive).
       * If no elements in range min-max are found in the list, the list's
contents are unchanged.
       * If an empty list is passed, the list remains empty.  Assume min < max.
       * @param listOfInts the list of Integers (list can be empty)
       * @param min the minimum value in the range
       * @param max the maximum value in the range
       * @return the given list with the range min-max removed
       */
      public static ArrayList<Integer> filterRange(ArrayList<Integer>
listOfInts, int min, int max)  {
            for (int i = 0; i < listOfInts.size(); i++) {
                  for (int j = min; j <= max; j++) {
                        if (listOfInts.get(i) == j) {
                              listOfInts.remove(i);
                              i--;
                              break;
                        }
                  }
            }
            return listOfInts;   // This return statement should be last
      }


      /**
       * Models/simulates the game of Bulgarian Solitaire.
       * @param numCards the number of cards to start with; n must be a
triangular number (a triangular
       * number is a number that can be written as the sum of the first n
positive integers).
       */
      public static void bulgarianSolitaire(int numCards)  {

            // Check if given number of cards is triangular
            int n = (int) Math.sqrt(2*numCards);
            if (n*(n+1)/2 != numCards)  {
                  System.out.println(numCards + " is not triangular");
                  return;
            }

            Random rand = new Random();

            ArrayList<Integer> finalSplits = new ArrayList<Integer>();
            for (int i = 1; i <= n; i++) { //make an array with card piles in
ascending order
                  finalSplits.add(i);
            }
```

```
            ArrayList<Integer> cardSplits = new ArrayList<Integer>();

            while (numCards > 0) { //split piles into random # of cards
                  if (numCards == 1) {
                        cardSplits.add(1);
                        break;
                  }
                  int pile = rand.nextInt(numCards - 1) + 1;
                  cardSplits.add(pile);
                  numCards -= pile;
            }

            System.out.print("Starting deck: ");
            for (int pile: cardSplits) { //print out beginning array
                  System.out.print(pile + " ");
            }
            System.out.println();

            int iterationCount = 0;
            while (cardSplits.containsAll(finalSplits) == false) { //while
cardSplits doesn't have cards in ascending order
                  int newPile = 0;
                  for (int i = 0; i < cardSplits.size(); i++) { //subtract 1
card from each pile
                        cardSplits.set(i, cardSplits.get(i) - 1);
                        newPile++;
                        if (cardSplits.get(i) == 0) { //if pile has 0 cards
remove it
                              cardSplits.remove(i);
                              i--;
                        }
                  }

                  cardSplits.add(newPile); //add the pile made up of the
removed cards
                  System.out.print("Iteration " + ++iterationCount + ": ");
                  for (int pile: cardSplits) { //print new pile config
                        System.out.print(pile + " ");
                  }
                  System.out.println();
            }

      }
}
```