CS 211, ALL SECTIONS

PROJECT 5

DUE WEDNESDAY, MAY 1ST AT MIDNIGHT

The objective of this project is to make use of the Java Collection framework to implement a contact list class along with a utility class which provide static (some recursive) methods to support the contact list class's functionality. The project will make use of inner classes and recursion while reinforcing skills involving working with generics.
Another objective of this project is the focus on the ability to write test cases for an existing class.

OVERVIEW:

1. Implement the class `Contact` which will be used to save basic information about a contact.
2. Implement a utility class `PhoneBookUtils` which includes static methods to be used by the `PhoneBook` class when manipulating the contact list.
3. Implement the class `PhoneBook`, which uses a *map* data structure to maintain a list of contacts, their emails and their phone numbers.
4. Implement the private inner class `PhoneBookIterator` which implements `Iterable`.
5. Students in the honors section will have an additional class to implement for full credit.
6. Write test cases to test each method in the four classes you have implemented.
7. Download and use the tester module to ensure that your program is correct.
8. Prepare the assignment for submission and submit it.

For this project, we will implement an electronic phone book. Like in previous projects, we will build from the ground up by making the phone book classes. We will build up to a tester program which populates the contact list (by reading from an input file) and implement the different methods to check the validity of the application. We will incorporate two more programming techniques into this project: maps (as data structures) and recursion.

We will use a `HashMap` (from the Collections Framework) to store contact information. If we just wanted a list, we could use `ArrayList`, but by using a map data structure we can maintain a mapping between a key and a value. The key would be a contact's name, while a value entry will contain other information related to the contact (email/phone). Remember that a map does not allow duplicate keys and therefore we get the guarantee that there is only one entry per contact in our data structure. We will also implement recursive algorithms for sorting the list of contacts and also when searching for a contact (recursive insertion sort / binary search).

Finally, in this project, you are expected to write your own test methods to test your code! We will provide you with a sample tester program which you can use for the purpose of testing your own methods. Make sure to write enough test cases for the different scenarios for each method. For instance, what would happen if we try to add a duplicate contact to the hash map? Or what would happen if we invoke the getContactList on an empty list? These border cases must be covered when you are testing your code. Be sure to test your code thoroughly.

RULES

- This project is an individual effort; the Honor Code applies
- You may import Java built-in libraries, since this is a project about Collections.
- You may add your own helper methods or data fields, but they should be private. You may add additional constructors which are public.
- You may NOT add any extra public methods other than those outlined in this document.
- Some methods provide specific implementation instructions, such as which methods must or cannot be used, or whether the method must be implemented recursively. These constraints may be verified manually.

1. CONTACT TASK:

(15pts) `public class Contact implements Comparable<Contact> :` This class maintains information for a single contact, namely the following 3 instance variables are defined: name, phone and email, all of type `String`. Note that this class will be used when defining an entry type for the hash map used in the `PhoneBook` class. Furthermore, this class implements `Comparable` which makes instances of this class to be comparable against others. This is useful when we need to call any of the generic methods in the Collection framework which requires a type that is comparable, such as the static method `Collections.sort` defined in the Collections utility class. Additionally, this class will provide the following public methods:

- A single parameterized constructor which receive 3 String parameters and use those values to initialize the 3 instance variables defined by the class.
- Accessor methods (get/set) for the instance variables as follows: `getName`, `getEmail`, `getPhone`, `setEmail`, and `setPhone`. Only a get method is needed for the name field (names cannot be changed after a contact is created), while both set and get methods are expected for both phone and email.
- `@Override public String toString()` : returns a string in the format "*NAME*, email: *EMAIL*, phone: *PHONE*.". Note the dot "." At the end of the line, for example:

```
> Contact c = new Contact("Arthur Moddy", "est@temp.com", "703-555-5555");
> System.out.println(c);
Arthur Moddy, email: est@temp.com, phone: 703-555-5555.
```

- `@Override public int compareTo(Contact c)` This method will return the result of the comparing the lowercase versions of the name portions of both contacts (a String itself is also a `Comparable`). More precisely, the method will convert the name of `this` contact and the name of `c`'s contact to lowercase and invoke the `compareTo()` method on the former, passing in the latter as an argument to the call. It will return an integer value which is the result of the two name strings comparison.

2. PHONEBOOKUTILS TASK:

(30*pts*) `public final class PhoneBookUtils :` this is a utility class, which means that we do not intend to instantiate it at all. Instead, we will use the static helper methods it provides while writing other classes. Examples of other utility classes are `java.lang.Math`, `java.util.Arrays`, or even the class we wrote in the first project. Utility classes such as this one are often declared `final` such that they cannot be subclassed. We will use it to define generic methods with the idea that we could reuse the same methods across a number of different applications.

In our phone book application, the utility class implements the following static methods:

- `public static <KeyType, ValueType> String mapToString(Map <KeyType, ValueType> map):` a generic method which receives a `Map` and returns a string containing all of the the map's "entries." It should use the `toString()` method of each map entry, and place a newline after each entry so that there is one entry per line.
- `public static <T extends Comparable<T>> String listToSortedList(List <T> list)` : a generic method which receives a `List` and returns a string containing the "entries" in the list in *sorted* order, separated by newlines so that there is one entry per line. The `toString()` method of each entry should be used to obtain a string representation. A side effect of the method is that the input list should become sorted. This method may make use of the static helper method `Collections.sort`.
- `public static <T extends Comparable<T>> void insertionSort(T [] arr, int i)` : a generic method which receives an array and sorts the array elements using the insertion sort algorithm. To sort the array, the method should intially be called with an `i` parameter of 0 or 1. This method must be *recursive* as follows: when it is called, it should be assumed that the first `i` elements of the array are already sorted, while the remaining elements may or may not be. The sorted portion of the array will always have at least one element, because any one element is always in sorted order in relation to itself. The recursive step will always place one additional element into sorted order by inserting it into the sorted section, in its proper sorted position, thus increasing the size of the sorted section by one element. Then, it will make a recursive call to sort the remaining elements. In otherwords, the recursive step works by assuming that subset arr[0..i-1] of the array is already sorted, and works on the arr[i..n] (which is the unsorted subset) by inserting first unsorted element (which is a[i]) into the sorted region and incrementing the value of `i` for each recursive call. The base case is reached once all elements have been placed into sorted order and there is no unsorted portion of the list remaining.
- `public static <T extends Comparable<T>> T binarySearch(T arr[], int begin, int end, T key)` : this generic method accepts an array and a key object returns the object contained in the array which matches the search key (using `compareTo()`) if found in the index range ≥ `begin` and < `end`, or null if key is not found in that index range. *You may assume that the input array is already in sorted order.* You may *not* call built-ins such as `Arrays.binarySearch()` or `Collections.binarySearch()` as a part of your solution, and you must implement the binary search algorithm recursively.

Here is a sample run for the use of the methods in this class:

```
> import java.util.*;
> Map<String, Integer> map = new HashMap<String, Integer>();
> map.put("x",1); map.put("y",2); map.put("z",3);
> map
{x=1, y=2, z=3}
> System.out.print(PhoneBookUtils.mapToString(map));
1
2
3
> Integer[] ints = { 3, 8, 5, 4, 1, 9, -2 };
> List<Integer> intList = new ArrayList<Integer>(Arrays.asList(ints));
> System.out.println(java.util.Arrays.toString(ints));
[3, 8, 5, 4, 1, 9, -2]
> System.out.println(intList);
[3, 8, 5, 4, 1, 9, -2]
```

```
> System.out.print(PhoneBookUtils.listToSortedList(intList));
-2
1
3
4
5
8
9
> System.out.println(intList);
[-2, 1, 3, 4, 5, 8, 9]
> PhoneBookUtils.insertionSort(ints, 0);
> System.out.println(java.util.Arrays.toString(ints));
[-2, 1, 3, 4, 5, 8, 9]
 > System.out.println(PhoneBookUtils.binarySearch(ints, 0, ints.length, 9));
9
> System.out.println(PhoneBookUtils.binarySearch(ints, 0, ints.length, 45));
null
> String[] strs = { "Hhh", "Ccc", "Fff", "Ttt", "Aaa", "Ddd" };
> List<String> strList = new ArrayList<String>(Arrays.asList(strs));
> System.out.println(java.util.Arrays.toString(strs));
[Hhh, Ccc, Fff, Ttt, Aaa, Ddd]
> System.out.println(strList);
[Hhh, Ccc, Fff, Ttt, Aaa, Ddd]
> System.out.print(PhoneBookUtils.listToSortedList(strList));
Aaa
Ccc
Ddd
Fff
Hhh
Ttt
> System.out.println(strList);
[Aaa, Ccc, Ddd, Fff, Hhh, Ttt]
> PhoneBookUtils.insertionSort(strs, 0);
> System.out.println(java.util.Arrays.toString(strs));
[Aaa, Ccc, Ddd, Fff, Hhh, Ttt]
 > System.out.println(PhoneBookUtils.binarySearch(strs, 0, strs.length, "Fff"));
Fff
> System.out.println(PhoneBookUtils.binarySearch(strs, 0, strs.length, "Bebe"));
null
```

---

3. PHONEBOOK TASK:

(*40pts*) `public class PhoneBook implements Iterable<Contact> :` Our phonebook class stores a list of contacts using a private `HashMap` data structure. As discussed in the *Overview* section, the hashmap would hold values ("entries") which are `Contact` objects, referenced by keys which are the lowercase version of the contact's name. This class provides the required functionality for maintaining the list of contacts: adding a new contact, updating a contact information, removing a contact, generating a list of all contacts, and sorting contacts by last name. This class can also read a list of contacts from an input file and can create an iterator to the list to be used when traversing the contact list. The PhoneBook class makes use of the available static methods which are available in the `PhoneBookUtils` class. Here is an example of how this class works:

```
> PhoneBook blackbook = new PhoneBook();
> blackbook.addContact("Lance Farmer", "sem.egestas@ctus.ca", "1-425-180-9073");
> blackbook.addContact("Baxter Cantu","arcu.iculis@iaculis.net","1-606-427-1676");
> blackbook.addContact("Arthur Moody", "est@temporerat.com", "1-170-451-6998");
> blackbook.addContact("Petra Wiley", "Morbi.m@duiquis.com", "1-357-588-7644");
> blackbook.addContact("Forrest Beach", "vulpute@Nuldum.co.uk", "1-117-275-2165");
> blackbook.getContactInfo("Arthur Moody")
"Arthur Moody, email: est@temporerat.com, phone: 1-170-451-6998."
> blackbook.getEmail("Forrest Beach")
"Forrest Beach: vulpute@Nuldum.co.uk."
> blackbook.getPhone("lAnCe FaRmEr") // note capitalization
"Lance Farmer: 1-425-180-9073."
```

```
> System.out.println(blackbook.getEmail("Donald Trump"));
null
> System.out.println(blackbook.getContactList());
Forrest Beach, email: vulputate@Nullainterdum.co.uk, phone: 1-117-275-2165.
Baxter Cantu, email: arcu.iaculis@iaculis.net, phone: 1-606-427-1676.
Petra Wiley, email: Morbi.metus@duiquis.com, phone: 1-357-588-7644.
Arthur Moody, email: est@temporerat.com, phone: 1-170-451-6998.
Lance Farmer, email: sem.egestas@urnanecluctus.ca, phone: 1-425-180-9073.
> System.out.println(blackbook.getSortedContactList())
Arthur Moody, email: est@temporerat.com, phone: 1-170-451-6998.
Baxter Cantu, email: arcu.iaculis@iaculis.net, phone: 1-606-427-1676.
Forrest Beach, email: vulputate@Nullainterdum.co.uk, phone: 1-117-275-2165.
Lance Farmer, email: sem.egestas@urnanecluctus.ca, phone: 1-425-180-9073.
Petra Wiley, email: Morbi.metus@duiquis.com, phone: 1-357-588-7644.
> System.out.println(blackbook.getSortedContactListAlt())
Arthur Moody, email: est@temporerat.com, phone: 1-170-451-6998.
Baxter Cantu, email: arcu.iaculis@iaculis.net, phone: 1-606-427-1676.
Forrest Beach, email: vulputate@Nullainterdum.co.uk, phone: 1-117-275-2165.
Lance Farmer, email: sem.egestas@urnanecluctus.ca, phone: 1-425-180-9073.
Petra Wiley, email: Morbi.metus@duiquis.com, phone: 1-357-588-7644.
> System.out.println(blackbook.searchContactList("Petra Wiley"))
Petra Wiley, email: Morbi.metus@duiquis.com, phone: 1-357-588-7644.
```

All methods which search for a contact by name (including all getters and setters) should be case insensitive, meaning that the method should first convert the name parameter to lowercase before searching for it in the hashmap. This class will have the following public methods:

- A default constructor.
- `public boolean fileToMap(String filename)`: reads contact information from an input file into the hash map. Returns true if the file was successfully read or false otherwise. The method should not throw any exceptions and must handle the possible exception (for example if the file cannot be opened). Entries come in groups of three lines: first a name, then an email address, then a phone number. There are no blank lines between entries (you do not need to check for this or to worry about whether the lines contain properly formated emails or phones). If there are not enough lines in the file for a final entry, then ignore the final one (*but still return true*). Here is an example of how data fields would appear in an input file (first-name last-name on one line, email on second line, and phone on the third file):

  Lance Farmer
  sem.egestas@urnanecluctus.ca
  1-425-180-9073
  Lawrence Garcia
  diam@amagna.ca
  1-665-672-6398
  Petra Williamson
  euismod.in@convallisligulaDonec.co.uk
  1-352-344-9237

  Tip: names contain spaces, so you will most likely want to read by line rather than by token; check the available methods in `Scanner`.
- `public boolean addContact(String name, String email, String phone)`: a `Contact` entry is created (refer to `Contact` class description in this document) and stored as an entry value in the `HashMap`, using the an all-lowercase version of the contact's name as the key value (for example, the contact `"George Mason"` would use the key `"george mason"`). Since duplicates are not allowed, this method returns false whenever a key matching the same name (all lower case) exists in the map, true otherwise.
- `public boolean deleteContact(String name)`: removes the element with a matching key from the map, returns true if the contact is deleted or false if the contact was not found in the map.
- `public Contact getContact(String name)` a getter method which finds the contact with the matching key and returns the `Contact` which was found, or null if not found.
- `public String getContactInfo(String name)` a getter method which finds the contact with the matching key and returns a text description of the contact using the `toString`() in `Contact`, or null if not found.
- `public String getEmail(String name)` and `public String getPhone(String name)`, getter methods which find the contact with the matching key and return a text description of the name/email of the contact, or null if not found. For either method, the method should return the description in the following format:`"NAME: EMAIL"` or `"NAME: PHONE"` .
- For updating a contact, we define `public boolean updateEmail(String name, String email)` and `public boolean updatePhone(String name, String phone)` : given a name and an updated field, these methods will find the

contact entry and update the given field (email / phone). These methods return true if an update is successful, or false if no matching entry is found in the map.

- `public String getContactList()` uses the helper static method `PhoneBookUtils.mapToString` method to get a string representation of the hash map, and returns it.
- `public String getSortedContactList()` which first converts the hash map "entries" into a `List` (for example type `ArrayList`) and then invokes the helper static method `PhoneBookUtils.listToSortedList` to get a string representation of a *sorted* entry list, and returns it. This method should not change the actual order of the entries within the hash map, hence the intermediate step. *Hint: applying the `.values()` method on the hashmap will return a list of the hashmap entry values.*
- `public String getSortedContactListAlt()` this method first converts the hash map "entries" into an array of type `Contact []` (using the `toArray(T[] a)` method provided by the collection), then it invokes the helper method `PhoneBookUtils.insertionSort` to get the array sorted, and finally it returns a string representation of the sorted array. The result of this method is identical to the `getSortedContactList()`, but differs by implementing a different technique to sort.
- `public String searchContactList(String name)`: this method will return a string representation of the chosen contact, which is to be found using the `PhoneBookUtils.BinarySearch` helper method. A suggested approach is to first convert the hash map "entries" into an array, then search for the contact within the array, finally turning it into a string and returning the result. Note that even if we do not know the email or phone, we can still use the search by creating a dummy object with a blank email and phone. We would create a dummy object with the name of the person we are looking for and "" for email and phone, and search for the dummy in the list. Since only the name field is used by the `compareTo` method in `Contact`, the dummy object will succeed in finding the real contact. `searchContactList` returns a string representation of the contact object if it is found or null otherwise.
- `@Override public Iterator<Contact> iterator()` this method will return an iterator which will allow iterating over the contacts in sorted order. A suggested approach is to first convert the contact list hash map into an `ArrayList` of "entry" values, then sort the list using the static method `Collections.sort`, and finally instantiate a new iterator object by passing it the sorted array list. The method will return the created iterator object, which iterates over the sorted "entry" values.

4. PhoneBookIterator Task:

(15*pts*) `public class PhoneBookIterator<E> implements Iterator<E>`: An iterator implementation in Java is just a class which implements the `Iterator` interface. Since we are going to use this class inside a `PhoneBook` we make this class an *inner class* (or *nested class*), which just means that we define it inside of the `PhoneBook` class. We may make inner classes public or private, but for this project we will use public so that we can see the implementation.

This iterator class will maintain two instance variables: a private `ArrayList` structure (which becomes the iterable object once the iterator is instantiated), and a private integer variable which keeps track of the current position of the iterator. It includes the following public methods:

- A single parameterized constructor which receives an `ArrayList<E>` through its parameter list and uses it to initialize its internal array list structure. The constructor method would also initialize the position variable to 0.
- `@Override public boolean hasNext():` returns true if there are more objects in the iterable structure, or false otherwise.
- `@Override public E next():` returns the next element to be read from the iterable structure, and increases position forward by 1. This method must throw an `NoSuchElementException`() exception when there is no next element to be read (position of iterator is beyond the size of the array list).

Here is a sample run to show how this class works:

```
> java.util.Iterator<Contact> it = blackbook.iterator();
> while (it.hasNext()) System.out.println(it.next());
Arthur Moody, email: est@temporerat.com, phone: 1-170-451-6998.
Baxter Cantu, email: arcu.iculis@iaculis.net, phone: 1-606-427-1676.
Forrest Beach, email: vulpute@Nuldum.co.uk, phone: 1-117-275-2165.
Lance Farmer, email: sem.egestas@ctus.ca, phone: 1-425-180-9073.
Petra Wiley, email: Morbi.m@duiquis.com, phone: 1-357-588-7644.
> it.next()
java.util.NoSuchElementException
      at PhoneBook$PhoneBookIterator.next(PhoneBook.java:169)
      at PhoneBook$PhoneBookIterator.next(PhoneBook.java:1)
```

HONORS SECTION:

If you are in the honors section, you must complete this part and it is worth 20 points of the project grade. If you are not in the honors section, you are welcome to attempt this but you do not need to complete it and it is not worth any points if you do.

This part of the project asks you to solve four problems using recursion (5 pts each). For each method, the method must be recursive in the sense that it includes invocation calls to itself. The four method should be placed in a class named `RecursionFun`.

`public static <T> void reverseArray(T[] arr, int begin, int end) :` a generic recursive method that reverses the portion of the input array starting at index `begin` and ending just before index `end`. Use recursion; do not use a loop.

`public static int arraySumExtra(int[] arr) :` a recursive method which will compute the sum of all numbers in the array, but with the largest element counted twice. So if the input is {1, 6, 2, 9, 3}, then the return value will be 1+6+2+(9+9)+3=30. Use recursion; do not use a loop.

`public static double addRecipocals(int n) :` a recursive method which takes an integer as a parameter and returns the sum of the first `n` reciprocals (the non-integer fractional values 1/1, 1/2, 1/3, etc). Use recursion; do not use a loop

`public String revString(String target) :` a recursive method that returns a String that is the reverse of the input String. Example: `System.out.println(recursiveObject.revString("George Mason University"));` would return a String equal to this: `"ytisrevinU nosaM egroeG"`.

TESTING:

a sample testing file is provided with tests for some methods, but you must write your own test cases for a majority of your public methods. A portion of your grade on each section will depend on whether you have written test cases for the supplied methods. Public methods behavior must adhere to the definition provided in this document. **Be sure to test your code thoroughly.**

SUBMISSION:

Submission instructions are as follows.

1. Let *xxx* be your lab section number, and let *yyyyyyyy* be your GMU userid. Create the directory `xxx_yyyyyyyy_P5/`
2. Place your files in the directory you've just created.
3. Create the file `ID.txt` in the format shown below, containing your name, userid, G#, lecture section and lab section, and add it to the directory.

   Full Name: Donald Knuth
   userID: dknuth
   G#: 00123456
   Lecture section: 004
   Lab section: 213

4. compress the folder and its contents into a .zip file, and upload the file to Blackboard.