Address Book

Warning: It is strongly advised that you type the code given in this example yourself. Don’t copy/paste it, most probably you will get unexpected errors that you have never seen. Some bugs are deliberately introduced as well to avoid copy-pasting. TAs will not cooperate with you in debugging such errors☺.

Problem

- We want to build an address book that is capable of storing name, address & phone number of a person.

- Address book provides functionality in the form of a JOptionPane based menu. The feature list includes
  - Add – to add a new person record
  - Delete – to delete an existing person record by name
  - Search – to search a person record by name
  - Exit – to exit from application

- The Address book should also support persistence for person records

Approach for Solving Problem

Building a small address book generally involves 3 steps. Let us briefly discuss each step and write a solution code for each step

Step1 – Make PersonInfo class

- First of all you need to store your desired information for each person. For this you can create a user-defined data type (i.e. a class). Make a class PersonInfo with name, address and phone number as its attributes.
- Write a parameterized constructor for this class.
- Write print method in Person class that displays one person record on a message dialog box.
The code for **PersonInfo** class is given below.

```java
import javax.swing.*;

class PersonInfo {

    String name;
    String address;
    String phoneNum;

    //parameterized constructor
    public PersonInfo(String n, String a, String p) {
        name = n;
        address = a;
        phoneNum = p;
    }

    //method for displaying person record on GUI
    public void print() {
        JOptionPane.showMessageDialog(null, "name: " + name + 
                                  "address:" + address + "phone no:" + phoneNum);
    }
}
```

**Note:** Not declaring attributes as `private` is a bad approach but we have done it to keep things simple here.

**Step 2 – Make AddressBook class**

- Take the example of daily life; generally address book is used to store more than one person records and we don’t know in advance how many records are going to be added into it.
- So, we need some data structure that can help us in storing more than one PersonInfo objects without concerning about its size.
- **ArrayList** can be used to achieve the above functionality
- Create a class Address Book with an ArrayList as its attribute. This arraylist will be used to store the information of different persons in the form of PersonInfo Objects. This class will also provide `addPerson`, `deletePerson` & `searchPerson` methods. These methods are used for adding new person records, deleting an existing person record by name and searching among existing person records by name respectively.
- Input/Output will be performed through JOptionPane.
The code for **AddressBook** class is

```java
import javax.swing.*;
import java.util.*;

class AddressBook {

    ArrayList persons;

    //constructor
    public AddressBook () {
        persons = new ArrayList();
    }

    //add new person record to arraylist after taking input
    public void addPerson( ) {
        String name = JOptionPane.showInputDialog("Enter name");
        String add = JOptionPane.showInputDialog("Enter address");
        String pNum = JOptionPane.showInputDialog("Enter phone no");

        //construct new person object
        PersonInfo p = new PersonInfo(name, add, pNum);

        //add the above PersonInfo object to arraylist
        persons.add(p);
    }

    //search person record by name by iterating over arraylist
    public void searchPerson (String n) {
        for (int i=0; i< persons.size(); i++) {
            PersonInfo p = (PersonInfo)persons.get(i);
            if ( n.equals(p.name) ) {
                p.print();
            }
        } // end for
    } // end searchPerson
```
//delete person record by name by iterating over arraylist
public void deletePerson (String n) {
    for (int i=0; i< persons.size(); i++) {
        PersonInfo p = (PersonInfo)persons.get(i);
        if ( n.equals(p.name) ) {
            persons.remove(i);
        }
    }
}

The **addperson** method first takes input for name, address and phone number and then construct a **PersonInfo** object by using the recently taken input values. Then the newly constructed object is added to the arraylist – **persons**.

The **searchPerson & deletePerson** methods are using the same methodology i.e. first they search the required record by name and than prints his/her detail or delete the record permanently from the **ArrayList**.

Both the methods are taking string argument, by using this they can perform their search or delete operation. We used **for** loop for iterating the whole **ArrayList**. By using the **size** method of **ArrayList**, we can control our loop as **ArrayList** indexes range starts from 0 to one less than size.

Notice that, inside loop we retrieve each **PersonInfo** object by using down casting operation. After that we compare each **PersonInfo** object’s name by the one passed to these methods using **equal** method since **Strings** are always being compared using **equal** method.

Inside **if** block of **searchPerson**, **print** method is called using **PersonInfo** object that will display person information on GUI. On the other hand, inside **if** block of **deletePerson** method, **remove** method of **ArrayList** class is called that is used to delete record from **persons** i.e. **ArrayList**.
Step 3 – Make Test class (driver program)

- This class will contain a main method and an object of AddressBook class.
- Build GUI based menu by using switch selection structure
- Call appropriate methods of AddressBook class

The code for Test class is

```java
import javax.swing.*;

class Test {
    Public static void main (String args[]) {
        AddressBook ab = new AddressBook();
        String input, s;
        int ch;
        while (true) {
            input = JOptionPane.showInputDialog("Enter 1 to add 
\nEnter 2 to Search \nEnter 3 to Delete\n\nEnter 4 to Exit");
            ch = Integer.parseInt(input);
            switch (ch) {
                case 1:
                    ab.addPerson();
                    break;
                case 2:
                    s = JOptionPane.showInputDialog("Enter name to search ");
                    ab.searchPerson(s);
                    break;
                case 3:
                    s = JOptionPane.showInputDialog("Enter name to delete ");
                    ab.deletePerson(s);
                    break;
                case 4:
                    System.exit(0);
            }
        }
    }
}
```
Note that we use *infinite* while loop that would never end or stop given that our program should only exit when user enters 4 i.e. exit option.

**Compile & Execute**

Compile all three classes and run Test class. Bravo, you successfully completed the all basic three steps. Enjoy! But the handout is not yet finished….. A lot of work is ahead😊.
Modification - Adding Persistence Functionality

Hopefully, your address book is giving you the required results except one i.e. persistence. You might have noticed that after adding some person records in the address book; if you exit form the program next time on re-executing address book all the previous records are no more available.

To overcome the above problem, we will modify our program so that on exiting/starting of address book, all the previously added records are available each time. To achieve this, we have to provide the persistence functionality. Currently, we will accomplish this task by saving person records in some text file.

Supporting simple persistence by any application requires handling of two scenarios. These are

- On start up of application – data (person records ) must be read from file
- On end/finish up of application – data (person records) must be saved in file

To support persistence, we have to handle the above mentioned scenarios

Scenario 1 – Start Up

- Establish a data channel with a file by using streams
- Start reading data (person records) from file line by line
- Construct PersonInfo objects from each line you have read
- Add those PersonInfo objects in arraylist persons.
- Close the stream with the file
- Perform these steps while application is loading up

We will read records from a text file named persons.txt. The person records will be present in the file in the following format.

```
Ali,defence,9201211
Usman,gulberg,5173940
Salman,LUMS,5272670
```

As you have seen, each person record is on a separate line. Person’s name, address & phone number is separated using comma (,).
We will modify our AddressBook.java by adding a new method loadPersons into it. This method will provide the implementation of all the steps. The method is shown below:

```java
public void loadPersons (){
    String tokens[] = null;
    String name, add, ph;

    try {
        FileReader fr = new FileReader("persons.txt");
        BufferedReader br = new BufferedReader(fr);
        String line = br.readLine();
        while ( line != null ) {
            tokens = line.split(",");
            name = tokens[0];
            add = tokens[1];
            ph = tokens[2];
            PersonInfo p = new PersonInfo(name, add, ph);
            persons.add(p);
            line = br.readLine();
        }
        br.close();
        fr.close();
    }catch(IOException ioEx){
        System.out.println(ioEx);
    }
}
```

- First, we have to connect with the text file in order to read line by line person records from it. This task is accomplished with the following lines of code

  ```java
  FileReader fr = new FileReader("persons.txt");
  BufferedReader br = new BufferedReader(fr);
  ```

- FileReader is a character based (node) stream that helps us in reading data in the form of characters. As we are using streams, so we have to import the java.io package in the AddressBook class.

- We passed the file name persons.txt to the constructor of the FileReader.
Next we add BufferedReader (filter stream) on top of the FileReader because BufferedReader facilitates reading data line by line. (As you can recall from the lecture that filter streams are attached on top of node streams). That’s why the constructor of BufferedReader is receiving the fr – the FileReader object.

The next line of code will read line from file by using readLine( ) method of BufferedReader and save it in a string variable called line.

```java
String line = br.readLine();
```

After that while loop starts. The condition of while loop is used to check whether the file is reached to end (returns null) or not. This loop is used to read whole file till the end. When end comes (null), this loop will finish.

```java
while (line != null)
```

Inside loop, the first step we performed is tokenizing the string. For this purpose, we have used split method of String class. This method returns substrings (tokens) according to the regular expression or delimiter passed to it.

```java
tokens = line.split(“,“);
```

The return type of this method is array of strings that’s why we have declared tokens as a String array in the beginning of this method as

```java
String tokens[];
```

For example, the line contains the following string

```
Ali,defence,9201211
```

Now by calling split(“,“) method on this string, this method will return back three substrings Ali defence and 9201211 because the delimiter we have passed to it is comma. The delimiter itself is not included in the substrings or tokens.

The next three lines of code are simple assignments statements. The tokens[0] contains the name of the person because the name is always in the beginning of the line, tokens[1] contains address of the person and tokens[2] contains the phone number of the person.

```java
name = tokens[0];
add  = tokens[1];
ph   = tokens[2];
```

The name, add and ph are of type String and are declared in the beginning of this method.

After that we have constructed the object of PersonInfo class by using parameterized constructor and passed all these strings to it.
PersonInfo p = new PersonInfo(name, add, ph);

- Afterward the PersonInfo object’s p is added to the arraylist i.e. persons.
  
  persons.add(p);

- The last step we have done inside loop is that we have again read a line from the file by using the readLine() method.

- By summarizing the task of while loop we can conclude that it reads the line from a file, tokenize that line into three substrings followed by constructing the PersonInfo object by using these tokens. And adding these objects to the arraylist. This process continues till the file reaches its end.

- The last step for reading information from the file is ordinary one – closing the streams, because files are external resources, so it’s better to close them as soon as possible.

- Also observe that we used try/catch block because using streams can result in raising exceptions that falls under the checked exceptions category – that needs mandatory handling.

- The last important step you have to perform is to call this method while loading up. The most appropriate place to call this method is from inside the constructor of AddressBook.java. So the constructor will now look like similar to the one given below:

```
 public AddressBook () {
    Persons = new ArrayList();
    loadPersons();
 }
```

... AddressBook.java ...

**Scenario 2 – End/Finish Up**

- Establish a data channel(stream) with a file by using streams
- Take out PersonInfo objects from ArrayList (persons)
- Build a string for each PersonInfo object by inserting commas (,) between name & address and address & phone number.
- Write the constructed string to the file
- Close the connection with file
- Perform these steps while exiting from address book.
Add another method `savePersons` into `AddressBook.java`. This method will provide the implementation of all the above mentioned steps. The method is shown below:

```java
public void savePersons() {
    try {
        PersonInfo p;
        String line;

        FileWriter fw = new FileWriter("persons.txt");
        PrintWriter pw = new PrintWriter(fw);

        for (int i = 0; i < persons.size(); i++) {
            p = (PersonInfo) persons.get(i);
            line = p.name + "," + p.address + "," + p.phoneNumber;

            // writes line to file (persons.txt)
            pw.println(line);
        }
        pw.flush();
        pw.close();
        fw.close();
    } catch (IOException ioEx) {
        System.out.println(ioEx);
    }
}
```

- As you can see, that we have opened the same file (persons.txt) again by using a set of streams.
- After that we have started `for` loop to iterate over arraylist as we did in `searchPerson` and `deletePerson` methods.
- Inside `for` loop body, we have taken out `PersonInfo` object and after type casting it we have assigned its reference to a `PersonInfo` type local variable `p`. This is achieved by the help of following line of code
  ```java
  p = (PersonInfo) persons.get(i);
  ```
- Next we build a string and insert commas between the `PersonInfo` attributes and assign the newly constructed string to string’s local variable `line` as shown in the following line of code.
line = p.name +","+ p.address +","+ p.phoneNum;

Note: Since, we haven’t declare PersonInfo attributes private, therefore we are able to directly access them inside AddressBook.java.

- The next step is to write the line representing one PersonInfo object’s information, to the file. This is done by using println method of PrintWriter as shown below

    pw.println(line);

After writing line to the file, the println method will move the cursor/control to the next line. That’s why each line is going to be written on separate line.

- The last step for saving information to the file is ordinary one – closing the streams but before that notice the code line that you have not seen/ performed while loading persons records from file. That is

    pw.flush();

The above line immediately flushes data by writing any buffered output/data to file. This step is necessary to perform or otherwise you will most probably lose some data for the reason that PrintWriter is a Buffered Stream and they have their own internal memory/storage capacity for efficiency reasons. Buffered Streams do not send the data until their memory is full.

- Also we have written this code inside try-catch block.

- The last important step you have to perform is to call this method before exiting from the address book. The most appropriate place to call this method is under case 4 (exit scenario) in Test.java. So the case 4 will now look like similar to the one given below:

```
......
case 4:
    ab.savePersons();
    System.exit(0);
......
```

Test.java

Compile & Execute

Now again after compiling all the classes, run the Test class. Initially we are assuming that out persons.txt file is empty, so our arraylist persons will be empty on the first start up of address book. Now add some records into it perform search or delete operations. Exit from the address book by choosing option 4. Check out the
persons.txt file. Don’t be getting surprised by seeing that it contains all the person records in the format exactly we have seen above.

Next time you will run the address book; all the records will be available to you. Perform the search or delete operation to verify that.

This example is the first step for building large assignments of this course. We hope that you enjoyed it 😊