ECE25100 Object Oriented Programming

Lab 6: ArrayLists, Wrapper Classes and Iterators

Description:
The purpose of this lab is to help you understand how to use ArrayLists, Wrapper classes and Iterators.

To get credit for the lab, you need to demonstrate to the student helper that you have completed all the requirements.

Question 1:
Step 1: It is important to understand the difference between **arrays** and **ArrayLists**. Recall that both arrays and **ArrayLists** can store multiple types of objects. Copy/paste the code below into a file called **ArrayListTestProgram1.java**.

```
import java.util.ArrayList;
public class ArrayListTestProgram1 {
    public static void main(String args[]) {
        int[] numbersArray = new int[10];
        ArrayList numbersArrayList = new ArrayList(10);

        System.out.println("The array looks like this: " +
         numbersArray);
         System.out.println("It has " + numbersArray.length + "
         elements in it");
         System.out.println("The 5th element in it is: " +
         numbersArray[4]);

         System.out.println("\nThe arrayList looks like this: " +
         numbersArrayList);
         System.out.println("It has " + numbersArrayList.size() + "
         elements in it");
         System.out.println("The 5th element in it is: " +
         numbersArrayList.get(4));
    }
}
```

Compile the code. Notice that variables holding **arrays** are declared in a different way than variables holding **ArrayLists**. Also, notice that **ArrayLists** use a **size()** method instead of a **length** attribute in order to determine the number of elements in them. Lastly, notice that we can access the particular elements in **ArrayLists** by using the **get()** method instead of the square brackets **[ ]**.

Run the code. Notice that:

- It generates an exception ... ignore this for now ...

- **ArrayLists** "look" different when printed.
- **ArrayLists** have a size of 0 when nothing is in them, while **arrays** have a fixed size according to the capacity specified when they are declared.
- The **ArrayList** is actually empty so we cannot get the 5th element from it, whereas **arrays** are never empty ... they are initialized with zeros upon creation.

There is nothing more to do on this part. Just make sure that you understand why we got an error for the ArrayList but not for the Array.

Step 2: Now we will try filling up the **ArrayList**. Add the following code after the **ArrayList** declaration:

```
numbersArray[0] = 56;
numbersArray[1] = 78;
numbersArray[2] = 90;
numbersArray[3] = 12;
numbersArray[4] = 23;

numbersArrayList.add(56);
numbersArrayList.add(78);
numbersArrayList.add(90);
numbersArrayList.add(12);
numbersArrayList.add(23);
```

Notice that we usually **add** to an **ArrayList** whereas with **arrays** we specifically set the elements at each indexed location. Compile the code. You should see a compiler "warning" that looks something like this:

```
Note: H:\ArrayListTestProgram1.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

We are getting this warning because we are adding **ints** to an **ArrayList**. By default, **ArrayLists** are meant to store objects, not primitives (e.g., **int**, **float**, **char** etc..). However, in JAVA version 5, adjustments were made to allow us to store primitives into the **ArrayLists**. The warning that we are receiving here is instructing us that we should have made it clear in our code that we want to store primitive **ints** in the **ArrayList**.

However, the code will still run. We can get rid of this warning by explicitly declaring that the **ArrayList** will hold integers. Change the declaration of the **ArrayList** so that it reads as follows:

```
ArrayList<Integer>   numbersArrayList = new ArrayList<Integer>(10);
```

Now re-compile. The warning will be gone. This code actually informs the compiler that the **ArrayList** will contain **Integer** objects, not **int** primitives. However, you may recall that **Integer** objects are wrapper classes for **int** primitives. Since version 5, JAVA now automatically converts the wrapper objects into primitives when you use them in an expression.

Step 3: An iterator in JAVA is used to loop through elements from a collection such as an **ArrayList**. It is often used as a replacement for the FOR loop. Consider the following test program:

```
import java.util.ArrayList;
import java.util.Iterator;

public class ArrayListTestProgram2 {
    public static void main(String args[]) {
        ArrayList<Integer>  numbersArrayList = new
         ArrayList<Integer>(
         java.util.Arrays.asList(1, 45, 23, 87, 89, 213, 54, 11, 76, 98,
         23, 5));

         System.out.println("The ArrayList looks like this beforehand:
         " + numbersArrayList);

         for (int num: numbersArrayList) {
             if (num%2 != 0)
                 numbersArrayList.remove(num);
         }

         System.out.println("The ArrayList looks like this afterwards:
         " + numbersArrayList);
    }
}
```

This code attempts to remove all of the odd numbers from the **ArrayList**. But, if you compile and run it, you will notice that JAVA generates a java.util.ConcurrentModificationException. This is because JAVA does not allow you to modify a collection (such as an **ArrayList**) while you are looping through it. So, we cannot simply go through and remove the odd numbers in this way. Of course, we could make a new **ArrayList** and then copy over the even numbers into it, but this requires a new object and is sometimes not what we really want to do.

Iterators are used for this purpose. Add the following line to your code and then add an **import java.util.Iterator;** to the top of your code:

```
Iterator numsIterator = numbersArrayList.iterator();
```

Now replace the **for** loop with a **while** loop that uses the iterator:

```
while (numsIterator.hasNext()) { ... }
```

This loop will repeat until the iterator has no more elements in it (i.e., until you have *iterated through* (or checked) all of the elements in the **ArrayList**). Complete the code. To get the next element from the iterator, you send the **next()** message to it. In this case, **next()** **Integer** object that is in the ArrayList. You still need to check to see whether or not it is odd. You will have to typecast the result of the **next()** method to an **Integer** in order to use it in a math expression. When you find an odd number, you can send the **remove()** message to the iterator to have that element removed. Remember, **remove()** does not take any parameters, it simply removes the last element that you just got from the iterator. Make sure that you test your code when done to see if it really did remove all of the odd numbers.

Question 2: Download a **Customer** class, a **Store** class and a test program called **StoreTestProgram.java**. Compile all three files and run the test program.

The **Store** class contains 4 incomplete methods which you must write as shown below. Upon completing the method test it by running the test case in **StoreTestProgram**. Here are the methods:

1. **recordPurchase(Customer c, float price)** – record the purchase from the customer, and add the customer in the store record.
2.
3. **getCustomersOfSex(char sex)** - returns a list of all Customers in the store that have the same sex as the one specified in the parameter. Use a **for (Customer c: customers)** loop to fill up a newly created **ArrayList** and return it.
4. **friendsFor(Customer c)** - returns a list of all Customers in the store that are possible friends for the one specified in the parameter. Customers will be considered "possible friends" if they have the same sex and are within 3 years of age one each other. Use an **iterator** on the **customers ArrayList** to fill up a newly created **ArrayList** and return it.
5. **removeBrokeCustomers()** - removes all Customers from the store that have less than $10. Use an **iterator** on the **customers ArrayList** to remove the appropriate customers. There should not be anything returned from this method.

Once you write your code, you should notice that there are 15 males, 11 females, 3 friends for Amie, 4 friends for Brad and 22 remaining non-broke Customers.