ECE25100 Object Oriented Programming

Lab 7: File I/O and Object Streams

Description:
The purpose of this tutorial is to help you understand how to **read** and **write files** in *text*, *binary* and *object* formats.

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

STEP 1: We will now add code to save customers to files.

a. Download **Customer**, **Store** and **StoreTestProgram** classes. Compile all three files and run the test program to make sure that it works. Also, create, save and compile the following test program:

```
public class CustomerTestProgram {
    public static void main(String args[]) {
        Customer c1 = new Customer("Amie", 14, 'F', 100);
        Customer c2 = new Customer("Brad", 15, 'M', 0);
    }
}
```

b. Recall that we can create a **FileOutputStream** quite easily that allows us to output bytes to a file as follows:

```
FileOutputStream out = new FileOutputStream("customer1.txt");
out.write('A');
out.close();
```

However, to output a **Customer** in this manner, we would need to break it down into bytes. You will notice that a **Customer** is more complex ... containing a **name**, **age**, **gender** and amount of **money**. How would we break down a **String**, an **int**, a **char** and a **float** into bytes ? It is not so straight forward. So, JAVA created **DataOutputStream** objects to do this for us. Recall that a **DataOutputStream** just "wraps around" a **FileOutputStream** like this:

```
DataOutputStream out = new DataOutputStream(new
                       FileOutputStream("customer1.txt"));
```

Now we can write out the components of the object as follows:

```
out.writeUTF(c1.getName()); // UTF is short for "Unicode Text
Format"
out.writeInt(c1.getAge());
etc...
```

Add such code to the **CustomerTestProgram** so that the first customer **Amie** is saved to a file called **customer1.txt** and **Brad** is saved to a file called **customer2.txt**. You will need to **import java.io.*;** and don't forget to close the files when you are done using the **close()** method. Compile your code. Likely, you will get some errors indicating that you need to handle **java.io.FileNotFoundException** and **java.io.IOException**. Recall that we can handle these using **try/catch** blocks. Place your file-related code in a **try** block and **catch** the exceptions, doing nothing for now when they are caught:

```
try {
            // ... your code here ...
    } catch (FileNotFoundException e) {
            // Do Nothing
    } catch (IOException e) {
            // Do Nothing
    }
```

Compile and run your code now and then look into the folder that contains your code and you should see the two files. Open them with windows notepad to see what they look like. Likely, they each look weird:

```
Amie   FBÈ
```

```
Brad   M
```

This is because the information is stored as ***binary*.** As it turns out, when you open up binary information in a text editor (like notepad), the only information that will appear somewhat normal are **Strings** and **chars** since their byte values correspond to values that the text editor understands to be text.

c. If we want to save many **Customer** objects to a file, we would need to duplicate the 5 or 6 lines of code that do the saving. It could be very wasteful. Instead of doing this in our test code, it is more efficient to make a **Customer** method to do this. Create the following method in the **Customer** class:

```
public void saveTo(java.io.DataOutputStream aFile) throws
java.io.IOException { ... }
```

Copy the code that does the writing of the **Customer's** information into this method. Note that you won't use variables **c1** and **c2** now but you can replace it with the keyword **this** ... or even better you can access the 4 instance variables directly now instead of using the get methods. Make sure NOT to open nor close the file in this method, that is the responsibility of the method that called this one. Assume that the **DataOutputStream** has already been created and was passed in.

d. As a result, the try block of our test code should now look like this ... make the appropriate changes:

```
try {
      DataOutputStream  out;
      out = new DataOutputStream(new
      FileOutputStream("customer1.txt"));
      c1.saveTo(out);
      out.close();

      out = new DataOutputStream(new
      FileOutputStream("customer2.txt"));
      c2.saveTo(out);
      out.close();
      } catch (FileNotFoundException e) {
        // Do Nothing
        } catch (IOException e) {
        // Do Nothing
        }
```

Compile your **Customer** and **CustomerTestProgram** classes and then run the test code. It should still work. The code is shorter now and more efficient.

e. How do we read the **Customers** back in again ? Well, we can simply use a **DataInputStream** instead. This time, we'll make a static method so that we don't need to have a **Customer** already in order to load one from a file. Create the following **static** method in the **Customer** class:

```
public static Customer readFrom(java.io.DataInputStream aFile)
throws java.io.IOException { ... }
```

Complete the method so that a **Customer** object is created and returned from the method. The code should make use of the **DataInputStream**'s **readUTF()**, **readInt()**, **readChar()** and **readFloat()** methods to read and set the data for the customer.

f. Test your code by adding the following to the end of the **CustomerTestProgram** and compiling/running it. You should see the correct values displayed that correspond to the original customer's data.

```
try {
    DataInputStream  in;
    in = new DataInputStream(new
FileInputStream("customer1.txt"));
    System.out.println(c1.readFrom(in));
    in.close();
    in = new DataInputStream(new
FileInputStream("customer2.txt"));
    System.out.println(c1.readFrom(in));
    in.close();
} catch (FileNotFoundException e) {
    // Do Nothing
} catch (IOException e) {
    // Do Nothing
}
```

STEP 2: Of course, if we want a separate file for each **Customer** like this, we could have moved the code for opening and closing the file into the **Customer**'s **saveTo()** method, simply passing the file name as follows:

```
try {
        c1.saveTo("customer1.txt");
        c2.saveTo("customer2.txt");
    } catch (FileNotFoundException e) {
        // Do Nothing
    } catch (IOException e) {
        // Do Nothing
    }
```

Then each time we called the **saveTo()** method, a new file would be created or the old one overwritten. If however, we wanted to save many customers to the same file, we could not do it this way. Let us save both customers **c1 and c2** to the a single file called **customers.txt**.

a. Alter the first **try** block in our test code as follows:

```
try {
       DataOutputStream out = new DataOutputStream(new
FileOutputStream("customers.txt"));
       c1.saveTo(out);
       c2.saveTo(out);
       out.close();
    } catch (FileNotFoundException e) {
           // Do Nothing
    } catch (IOException e) {
           // Do Nothing
    }
```

Compile and run the code. It should work ... and here is what should appear in the text file:

```
Amie    FBÈ
Brad    M
```

Alter the second **try** block in our test code as follows:

```
try {
       DataInputStream in = new DataInputStream(new
       FileInputStream("customers.txt"));
       System.out.println(Customer.readFrom(in));
       System.out.println(Customer.readFrom(in));
       in.close();

    } catch (FileNotFoundException e){
       // Do Nothing
    } catch (IOException e){
```

```
            // Do Nothing
        }
```

Compile and run the code. It should work, showing the proper customers read back in.

b. Now we will save multiple customers to a common **store.txt** file. Write the following similar method now in the **Store** class:

```
public void saveTo(java.io.DataOutputStream aFile) throws
java.io.IOException { ... }
```

Complete this method so that it loops through the **Customer** objects and saves each one of them to the file. Make sure that you call your **saveTo()** method in the **Customer** class for each customer.

Test your code by adding the following to the end of the **StoreTestProgram** file's **main** method:

```
try {
        DataOutputStream out = new DataOutputStream(new
        FileOutputStream("store.txt"));
        walmart.saveTo(out);
        out.close();
    }catch (FileNotFoundException e){
                // Do Nothing
         }catch (IOException e){
                // Do Nothing
         }
```

Don't forget to **import java.io.*;** and then compile and run the code. Look at the resulting **store.txt** file. Did it work ?

c. Go to the **Store** class and write method below to read in the customers again. Notice that the code to actually read in the **Customer** object's data is missing. Write this code using the **readFrom()** method that you wrote in the **Customer** class.

```
public static Store readFrom(java.io.DataInputStream aFile)
throws java.io.IOException {
    Store s = new Store("?");
    while (aFile.available() > 0) {
        // Now read in a customer from the file and add him/her
        //to the store
    }
    return s;
}
```

Test your code by adding the following to the end of the **StoreTestProgram** file's **main** method:

```
try {
    DataInputStream in = new DataInputStream(new
    FileInputStream("store.txt"));
    walmart = Store.readFrom(in);
```

```
        in.close();
    } catch (FileNotFoundException e) {
        // Do Nothing
    } catch (IOException e) {
        // Do Nothing
    }
    System.out.println("Here are the customers from the file's Store
    object:\n");
    walmart.listCustomers();
```

Compile and run the test code. If your code was written properly, you should see the customers with names beginning from A to Z listed twice ... the first list represents data from the original object and the second list represents data that was read in from the file. The two lists should be identical.

STEP 3: Now although we can save and load the **Store** as well as **Customers**, it would be nice to save everything as text so that we can browse through it with a text editor. Lets do this now.

a. Go through all of your code in **Store**, **Customer**, and **StoreTestProgram** and do the following:

   i. replace **DataOutputStream** and **FileOutputStream** with **PrintWriter** and **FileWriter**, respectively.
   ii. replace **DataInputStream** and **FileInputStream** with **BufferedReader** and **FileReader**, respectively.
   iii. replace all of the **writeXXX()** code with **println()**.
   iv. replace all of the **readXXX()** code with **readLine()**. You will need to convert the incoming **age**, **gender** and **money** strings into **int**, **char** and **float** types. Do you remember how to do this ? We can say **Integer.parseInt(aString)** to an incoming string to convert it to an **int**. A similar approach can be used for **floats**. For chars, we simply do a **charAt(0)** to get the first **char** of the incoming string.
   v. There is no method called **available()** for **BufferedReader** objects. Instead, there is a method called **ready()** which returns a boolean indicating whether or not the buffer is able to be read. So, replace **aFile.available() > 0** with **aFile.ready()**.

b. Compile all your code and then run the **StoreTestProgram** ... it should work as before. However, open up the **store.txt** file to see what it looks like. It should look like this:

```
Amie
14
F
100.0
Brad
15
M
0.0
Cory
10
M
100.0
Dave
...
```

c. Modify your code so that a blank line separates the customers in the file as shown below. Make sure that your code still reads in the data properly:

```
Amie
14
F
100.0

Brad
15
M
0.0

Cory
10
M
100.0

Dave
...
```

STEP 4: Now see if you can save the customers into the **store.txt** file (and read them back in again) with this file format:

```
Amie,14,F,100.0
Brad,15,M,0.0
Cory,10,M,100.0
Dave,5,M,48.0
...
```

To read it in again, you will want to use the **split()** method from the **String** class. Recall an example of how to use it:

```
String s1 = "Bob,Tumacter,49,M,false";

String[] tokens = s1.split(",");
for(String token: tokens)
    System.out.println(token);
```

The code above will produce the following output:

```
Bob
Tumacter
49
M
false
```

Compile your code and test it by re-writing the file using the **StoreTestProgram**, viewing the **store.txt** file in Notepad to make sure that it saved in the correct format, and then re-reading in the store by running the **StoreReadTestProgram**.

STEP 5: Finally, we will save and then load the **Store** as an object (as opposed to binary and text as we have done). Recall that to save as whole objects, we simply need to implement the **Serializable** interface.

a. Add **implements java.io.Serializable** to your class definitions of **Customer** and **Store**. Re-compile both of those classes.

b. Add the following to the bottom of your **StoreTestProgram** so that it saves the **Store** using an **ObjectOutputStream** using the following lines:

```
try {
    ObjectOutputStream  objOut = new ObjectOutputStream(new
FileOutputStream("store2.txt"));
    objOut.writeObject(walmart);
    objOut.close();
} catch(IOException e) {
    // Do Nothing
}
```

You DO NOT need to add any methods in the **Customer** or **Store** classes! Just recompile the test program and run it. Look at the **store2.txt** file produced. It looks a little weird. In fact, it is a binary format which contains information about the class.

c. Add the following to the bottom of your **StoreTestProgram** so that it reads in the **Store** using an **ObjectInputStream** using the following lines:

```
try {
    ObjectInputStream  objIn = new ObjectInputStream(new
    FileInputStream("store2.txt"));
    walmart = (Store)objIn.readObject();
    objIn.close();
} catch(IOException e) {
    // Do Nothing
}
System.out.println("Here are the customers from the file's Store
object:\n");
walmart.listCustomers();
```

Try recompiling it. JAVA will tell you that you need to also now catch a **ClassNotFoundException**. That is because JAVA will try to reconstruct the **Store** objects itself upon reading it from the file and there may be problems creating objects if the file is corrupt, for example. You will need to add another **catch** block to catch this error. Now recompile and run the code.

d. Ok ... so it does not seem very *impressive* ... does it ? Lets look at how powerful this object saving/loading actually is. Go to your **Customer** and **Store** classes and comment out the **saveTo()** and **readFrom()** methods, then recompile the classes. Also, comment out the try/catch blocks in your StoreTestProgram that were from our PrintWriter and BufferedReader testing. Run the code. Your code will STILL work!! That is because these methods are no longer used, of course. So, when writing whole objects to a file, you do not need to do anything more that

simply implement the **Serializable** interface. The **readObject()** and **writeObject()** methods are available for free! Now do you see the benefits ?

e. Everything is not as "peachy" as it sounds though. Go to the **Customer** class and change the **money** variable to a **double** instead of **float**. Change the **getMoney()** method to return a **double** as well and the **setMoney()** method to take a **double** instead of **float**. Recompile this **Customer** class and the **Store** class. Now go to the **StoreTestProgram** and comment out the first **try/catch** block that saves the object to a file. There should only be one **try** block remaining in the test code. Run the code. It should run ok, but in fact, there was a problem. In the **catch** block for the **IOException**, add this line, then re-compile and run again :

```
System.out.println(e.getMessage());
```

Scroll up a bit and notice that the following error is displayed:

```
Customer; local class incompatible: stream classdesc
serialVersionUID = -6999499253118278227, local class
serialVersionUID = 6530349841534781369
```

As it turns out, the way the **Customer** and **Store** objects are stored in the file is dependent on the class definition itself. Since we changed the **Customer** class definition, we can no longer read in the **store.txt** file which is based on the previous version of the **Customer** class.

JAVA "*stamps*" each class with an ID called a ***serialVersionUID***. This changes each time the class is modified in some way such as modifying or deleting instance variables and other methods etc...

Now uncomment the 2nd last **try/catch** block in the **StoreTestProgram** so that the **store2.txt** file is re-written to the file using the new **Customer** class definition. The program should now be able to read it back in again without problems. Try it.

Do you see the *disadvantage* of using **ObjectStreams** ? Once you save the data, you need to keep the classes unchanged if you want to be able to read it back in. Of course, it is easiest to keep a copy of the code that was able to read the object back in. Later you can write programs that read it in and convert it to the new format. But it can really be a hassle. So, be aware of these issues when saving as objects.