**Lab 2 – Inheritance, Polymorphism & Data Streams**  
   
In this lab you will write a set of simple Java interfaces and classes that use inheritance and polymorphism. You will also write code that uses data (text and object) streams.

Like last time, you are to continue doing “***Pair Programming***” with your assigned partner. Each pair must work with one set of files.

***Objectives:***

* to implement and document interrelated interfaces and classes
* to use polymorphism in a program
* to make use of several types of input and output streams
* to read and write files from within a program
* to create a set of independent classes that interact with each other

***Problem 1: Workers Hierarchy***

***Part 1.1: Implement a class hierarchy***  
Use the following UML class diagram to implement the indicated interfaces and classes in Java. Unless the word *<<interface>>* appears, the entity is either a class or *abstract* class. Start by creating a ***worker*** packageinside your ***JavaPackages*** directory on your home(which will make the package accessible in later labs) andplace all the interfaces and classes inside that package. ***<<interface>>* Worker**   
                                  --------------------                               +getName():String

+equals(otherObject:Object):boolean

+toString():String  
                                       ^   
             //implements  | //extends   
                -----------------------|--------------------   
                |                                          |   
           ***<<abstract class>>* HourlyWorker**       ***<<interface>>* Employee** -----------------------------------------------------  
           -hours:int                            +getMonthlyPay():double   
           +getHours():int                        ^  
           +addHours(hours:int):void                  |   
           +resetHours():void                        |   
           ^ |//implements   
                |                                      |   
                |//extends              ---------------|-----------   
      ----------|------------------     |                        |   
      |                           |     |                        |   
**Volunteer                HourlyEmployee               SalariedEmployee**   
                           --------------               ----------------   
                           -hourlyPay:double           -monthlySalary:double   
                           +getHourlyPay():double      +setMonthlyPay(pay:double):void  
                           +setHourlyPay(pay:double):void   
   
Note the following clarifications and requirements:

* Do not add or remove methods or fields to or from any of the classes and interfaces
* Interface Worker must extend Serializable from java.io.\*
* The classes and/or interfaces where the name field belongs are not indicated; you need to decide where it *best* belongs in order to reduce code redundancy as much as possible. It may not be possible to put it in only one place
* All classes must override the equals method so that it compares workers by name; two workers are considered equal if they have the same name, even if they are not the same kind of *Worker*. If the other object is not some kind of *Worker*, the equals method should return *false*. Since this comparison is based on the name field, the equals method should be housed in the same class(es) containing the name field.
* Class HourlyWorker should be an *abstract* class because it does not override abstract toString method inherited from *Worker*. This methodshould be implemented in the subclasses; it should simply return the type of Worker subclass (i.e., return the String "*Volunteer*", "*HourlyEmployee*" or "*SalariedEmployee*", respectively)
* Field hours in HourlyWorker  represents the number of hours worked so far during the current month
* You also need to decide if any of the methods requires a precondition; if so, they should throw an appropriate exception when the precondition is not met

Supply appropriate constructors for all regular and abstract classes. Make sure to include different constructors to allow users to create instances with and without user-supplied values for all instance fields. Discuss your choices with your lab partner and come to agreement. Add appropriate Javadoc comments to all interfaces and classes. *Do this as you write your code NOT after you are done!* Work on the interfaces and classes until all have the indicated inheritance relationships and there are no compiler errors. Have the lab instructor or a TA review your work before you proceed.    
   
***Part 1.2: Testing the class hierarchy***  
Two complete JUnit test classes *VolunteerTest* and *WorkerPolymorphismTest* are provided; keep them inside your lab folder – ***do not move******them*** *to* your *JavaPackages* folder or the *worker* package folder.

Class *VolunteerTest* tests the constructors and all the methods of the *Volunteer* class; you can use it as soon as you have the *Volunteer* class and its superclasses/interfaces implemented. Note that the instance variable here is type *Volunteer* rather than *Worker*; in this case, using the specific variable type allows us to avoid casting before every method call.

Class *WorkerPolymorphismTest* determines that objects of the three regular classes are instances of the appropriate classes and interfaces and that they are not instances of any of the other classes or interfaces. You can use this test as soon as you have all six classes and interfaces compiling without errors. Note that it is necessary to use the type *Worker* for the local variables here to permit all the type comparisons; note that we also (perhaps unnecessarily) test to see that each object is an instance of its own class.

Use the *VolunteerTest* class as a model to create test the *HourlyEmployee* and *SalariedEmployee* classes. For each class, you will need to test its constructor(s) and all methods that pertain to it. In particular, you will need to add test methods for the methods that handle pay in these classes. Methods in the *HourlyEmployee* class that are implemented in the *HourlyWorker* class can use essentially the same tests as those for the *Volunteer* class.

When you are finished with all four tests, show them to the TA or lab instructor. Note that you must pass all test cases in the four test classes and you must show that the tests you designed are sufficiently thorough. (If you discover that the provided tests are insufficiently thorough, please point that out to the instructor right away.)

NOTE: We will return to these classes in the next lab, so it is important that you complete them by then

***Problem 2: Exercises in Data & Object Streams***

In this problem, you will first complete a Java class called CityNamesStream, which should contain the following three methods:

* *input2TextFile*: reads input from the console (System.in) and writes it to a standard text file
* *textFile2ObjectFile*: reads a text file, stores its data in a List object and writes the List as a single object to an object file
* *objectFile2Output*: reads a List object from an object file and displays its contents to the console (System.out).

An ***incomplete*** CityNamesStream class is provided to you in your lab folder. You are also given an ***incomplete*** driver class called CityNamesStreamDriver to allow you to test your methods in class CityNamesStream.

All classes and methods must be fully documented using Javadoc.

***Part 2.1:*** ***Copying text from the keyboard to a text file***

Class CityNamesStream contains the shell and Javadoc comments for method input2TextFile. You must complete this method according to the instructions below.

The method takes a String parameter textFileName. It first creates a Scanner object wrapped around System.in and a PrintWriter object wrapped around a File object to write to parameter textFileName. The method then reads lines of text (city names) from the console using the Scanner object and prints those lines (except for the last line) using the PrintWriter object. The last input line is an “end-of-input-indicator” and will consist of just the '.' character; compare the input line to the String "." using the equals method. When the method reads this last line, it does not write it to textFileName; instead, it closes the scanner and returns.

You can verify that this method is working correctly by running the accompanying driver program and viewing the text file it produces in a text editor; make sure that it has line breaks ***exactly and only*** where you typed them.

***Part 2.2: Copying text from a text file to a List and writing the list to an object file***

The second method is called textFile2ObjectFile; you can use method input2TextFile as a model.

This method takes two String parameters: a text file name and an object file name. It creates a new empty List <String> (such as an ArrayList), a Scanner object wrapped around a File object to read from the text file parameter, and an ObjectOutputStream wrapped around a FileOutputStream to write to the object file parameter. The method uses the scanner to read lines of text, appending each to the List, until it encounters the end of file (when the hasNextLine returns false); it then writes the List object to the ObjectOutputStream, closes the stream and scanner and returns.

In order to verify that this method is working correctly, you’ll need to expand the run method in the driver program to accomplish the following (Part 2 in the method’s Javadoc): *Get the name of an object file from the user, and call method textFile2ObjectFile on the same CityNameStreams instance passing the existing text file name and object file name as parameters*.

If your method is working correctly, it will generate an object file, but you will not be able to see the contents of the object file; you will complete your testing of this part in the next one.

***Part 2.3: Reading a List from an object file and writing the result to standard output.***

The third method should be called objectFile2Output; use the other two methods as templates.

This method takes the name of an existing object file as a String parameter and creates an ObjectInputStream wrapped around a FileInputStream for the parameter. It then reads the List from the stream and casts it properly, which might throw a ClassNotFoundException. The method then displays the elements of the List to the console using System.out, closes the stream, and returns.

In order to verify that this method is working correctly you’ll need to expand the run method in the driver program to accomplish the following (Part 3 in the method’s Javadoc): *Call method objectFile2Output on the same CityNameStreams instance passing the existing object file name as a parameter.*

Verify that ***Parts 2.2*** and ***2.3*** are working correctly by running the completed driver class. Output from ***Part 2.3*** should exactly match the original input from ***Part 2.1***, (except that the last line with only the period character is missing).