Lab 2 – Inheritance, Polymorphism & Data Streams

In this lab you will work on two separate problems. For the first problem, you will write a set of Java interfaces and classes that use inheritance and polymorphism. The second problem will require writing 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:

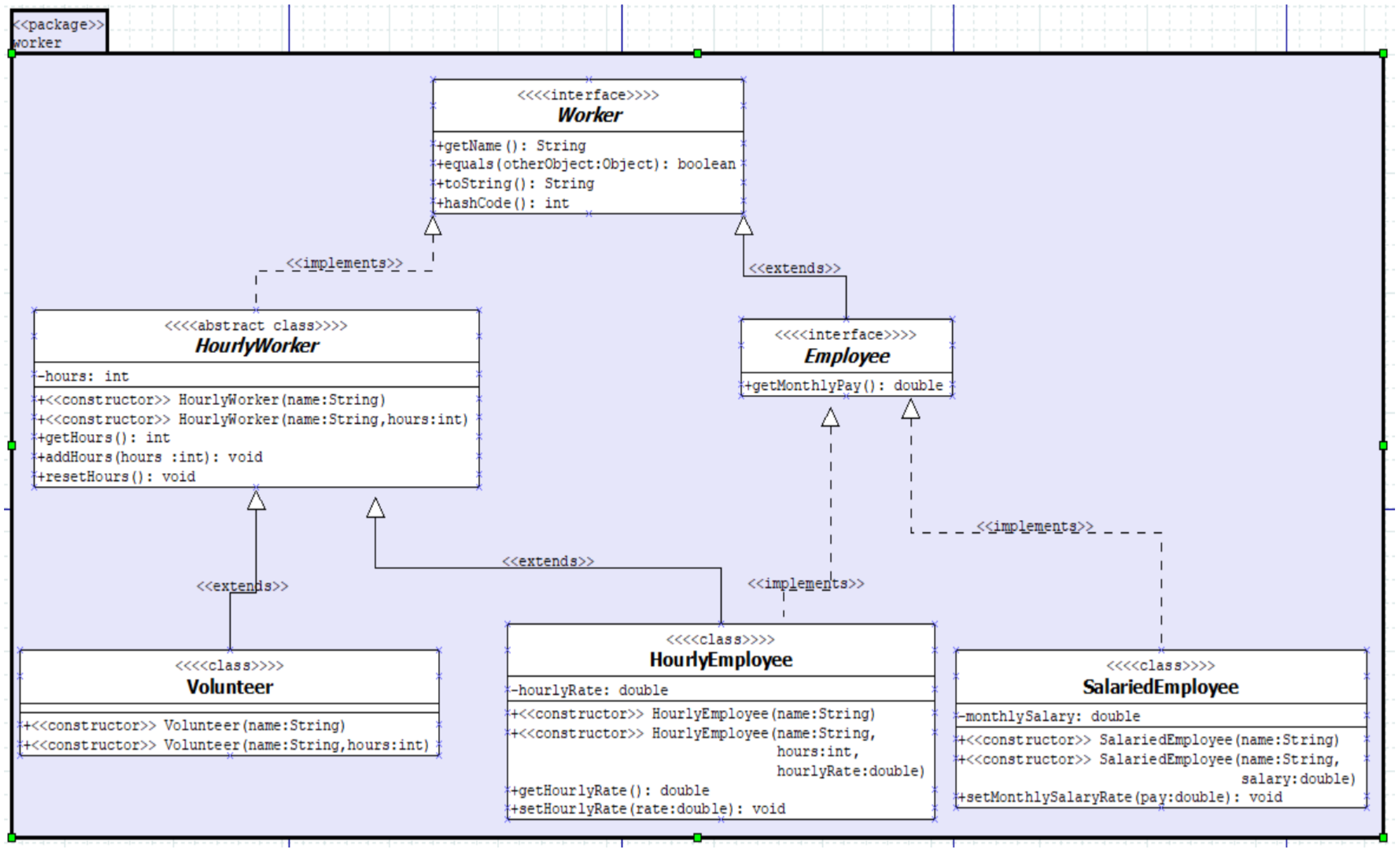
- to implement and document interrelated interfaces and classes
- to use polymorphism in a program
- to make use of several types of input and output streams
- to read and write files from within a program
- to create a set of independent classes that interact with each other

PS: Starting this lab, I plan to include a tests folder named something like ***Lab02Tests_forTAs*** (containing .class test files) inside your lab download. Please ignore this folder and keep it where it is.

Problem 1: Inheritance & Polymorphism: The worker Hierarchy

Part 1.1: Implementing and testing a class hierarchy

Use the following UML class diagram to implement all shown Java interfaces and classes inside a package called ***worker***. Start by creating a ***worker*** package inside your ***JavaPackages*** on your home (which will make the package accessible in later labs) and place all the interfaces and classes you create inside that package.



<<package>>
worker
<<<<interface>>>>
Worker
+getName(): String
+equals(otherObject:Object): boolean
+toString(): String
+hashCode(): int
<<implements>>
<<extends>>
<<<<abstract class>>>>
HourlyWorker
-hours: int
+<<constructor>> HourlyWorker(name:String)
+<<constructor>> HourlyWorker(name:String,hours:int)
+getHours(): int
+addHours(hours :int): void
+resetHours(): void
<<<<interface>>>>
Employee
+getMonthlyPay(): double
<<implements>>
<<extends>>
<<extends>>
<<implements>>
<<<<class>>>>
Volunteer
+<<constructor>> Volunteer(name:String)
+<<constructor>> Volunteer(name:String,hours:int)
<<<<class>>>>
HourlyEmployee
-hourlyRate: double
+<<constructor>> HourlyEmployee(name:String)
+<<constructor>> HourlyEmployee(name:String, hours:int, hourlyRate:double)
+getHourlyRate(): double
+setHourlyRate(rate:double): void
<<<<class>>>>
SalariedEmployee
-monthlySalary: double
+<<constructor>> SalariedEmployee(name:String)
+<<constructor>> SalariedEmployee(name:String, salary:double)
+setMonthlySalaryRate(pay:double): void

You are on your own for the most part, BUT please note the following important requirements and expectations:

[1] You must match all names exactly as shown in the diagram above (spelling and case). Do not include additional methods or fields to any of the classes and interfaces.

[2] Interface ***Worker*** must extend `Serializable` from `java.io.*`

[3] Classes where the ***name*** instance field belongs are not indicated; you need to figure out where it *best* belongs in order to reduce redundancy as much as possible (in other words, always put an instance field that is common to all subclasses once and only once in a superclass). NOTE that it will NOT be possible to put it in only one class–you will need at least two classes.

[4] Your hierarchy must override method ***public boolean equals(Object otherObject)*** to compare workers for equality by *name*; two workers are considered equal if they have the same *name*, even if they are not the same type of *Worker*. If *otherObject* is not an `instanceof` *Worker*, *equals* should return *false*. Since this comparison is based on the *name* field, your *equals* method should reside in the same classes containing the *name* field.

[5] Your hierarchy must also override method ***public int hashCode()*** to allow us to store such objects in data structures that use hashing (like HashMap or HashSet)–more on this topic later in the semester. For now, simply include the following method in the same classes containing the *name* field.

```
public int hashCode(){
 return this.getName().hashCode();
}
```

[6] Class ***HourlyWorker*** is an `abstract` class because it does not override `abstract` method *toString* inherited from *Worker*. Instead, this method is implemented in the leaf subclasses (i.e., classes at the lowest level in the hierarchy) by simply returning the type of *Worker* subclass–in other words, return String "*Volunteer*", "*HourlyEmployee*" or "*SalariedEmployee*", respectively.

[7] Field ***hours*** in ***HourlyWorker*** represents the number of hours worked so far during the current month.

[8] Keep an eye for methods requiring some sort of a precondition; such methods should throw an appropriate exception when the precondition is not met.

[9] As we did last time, you should be testing your class implementations AS you are building them not when you are all done. Start by implementing classes and interfaces needed to complete class *Volunteer* and use the supplied JUnit test class, *VolunteerTest*, to test your implementation. For the other classes, build them along with their test classes simultaneously, using class *VolunteerTest* as a model. See *Part 1.2* for more details.

Add appropriate Javadoc comments to all interfaces and classes. *Do this as you write your code NOT after you are done*! Do not forget to use *{@inheritDoc}* when need be. Once you have created all interfaces and classes along with the indicated inheritance relationships and there are no compiler errors, ask a TA to review your work before you proceed.

Part 1.2: Testing the class hierarchy

Two *****complete***** JUnit test classes *VolunteerTest* and *WorkerHierarchyStructureTest* are provided; keep them inside your lab folder–***do not move them*** to your *JavaPackages* folder or the *worker* package.

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 variables used here are of type *Volunteer* rather than *Worker*; this allows us to avoid casting before making calls to methods defined outside *Worker*.

Class *WorkerHierarchyStructureTest* tests that your hierarchy is accurately created to match the class diagram above. It also checks that objects of the three leaf classes (at the bottom) are indeed instances of the expected classes and interfaces in the hierarchy and not of others. You can run this test class as soon as you have all six classes and interfaces compiling without errors. Note that it is necessary here to use local variables of type *Worker* to permit the different `instanceof` comparisons.

Use class *VolunteerTest* as a model to create test classes for *HourlyEmployee* and *SalariedEmployee*. For each class, you will need to test its constructor(s) and all methods that pertain particularly to that (sub)class. In particular, you are expected to include tests for methods that handle pay in these classes. *HourlyEmployee* methods that are implemented in the *HourlyWorker* class can essentially reuse the corresponding tests in 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 asap.

Problem 2: Exercises in Data & Object Streams

Related class exercises that you'll find handy: *JavaIO.zip*.

In this problem, you will complete a Java class called ***CityNamesStream***, which contains the following three methods:

[1] *input2TextFile*: reads input from the console (`System.in`) and writes it to a text file

[2] *textFile2ObjectFile*: reads a text file, stores its data in a `List` Java object (such as an `ArrayList`) and writes the `List` <u>as a single object</u> to an object file

[3] *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 inside your lab folder. You are also given an ***incomplete*** driver class called *CityNamesStreamDriver* to allow you to interactively check your methods in class *CityNamesStream*. <u>Keep both classes inside your lab folder</u>.

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` (in other words, a `Scanner` object that takes `System.in` as a parameter) 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 and that the '.' character does not appear.

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 to complete this one.

This method takes two `String` parameters: a text file name and an object file name. It creates a new *empty* `List<String>`, 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.

To verify that this method is working correctly, you will need to expand the *run* method in the driver program to accomplish the following (Step 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. Try to open this file in a text editor; because it is an object file, it will look weird but take a close look at its contents–it should show the city names you input earlier. 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 to complete it.

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 will need to expand the *run* method in the driver program to accomplish the following (Step 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 complete 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).