CSCE 156 – Computer Science II

Lab 5.0 - Inheritance

Prior to Lab

1. Review this laboratory handout prior to lab.
2. Read the following tutorial on inheritance in Java
   http://download.oracle.com/javase/tutorial/java/IandI/subclasses.html
3. Read about abstract methods and abstract classes in Java
   http://download.oracle.com/javase/tutorial/java/IandI/abstract.html
4. Read about interfaces in Java:
   http://download.oracle.com/javase/tutorial/java/IandI/createinterface.html

Lab Objectives & Topics

Following the lab, you should be able to:

- Understand Inheritance and design classes and subclasses in Java
- Understand and use interfaces and abstract classes in Java
- Understand and use the `implements` and `extends` keywords

Peer Programming Pair-Up

To encourage collaboration and a team environment, labs will be structured in a *pair programming* setup. At the start of each lab, you will be randomly paired up with

another student (conflicts such as absences will be dealt with by the lab instructor). One of you will be designated the *driver* and the other the *navigator*.

The navigator will be responsible for reading the instructions and telling the driver what to do next. The driver will be in charge of the keyboard and workstation. Both driver and navigator are responsible for suggesting fixes and solutions together. Neither the navigator nor the driver is "in charge." Beyond your immediate pairing, you are encouraged to help and interact and with other pairs in the lab.

Each week you should alternate: if you were a driver last week, be a navigator next, etc. Resolve any issues (you were both drivers last week) within your pair. Ask the lab instructor to resolve issues only when you cannot come to a consensus.

Because of the peer programming setup of labs, it is absolutely essential that you complete any pre-lab activities and familiarize yourself with the handouts prior to coming to lab. Failure to do so will negatively impact your ability to collaborate and work with others which may mean that you will not be able to complete the lab.

Getting Started

Clone the project code for this lab from GitHub in Eclipse using the URL, `https://github.com/cbourke/CSCE156-Lab05`. Refer to Lab 1.0 for instructions on how to clone a project from GitHub.

Inheritance

Object Oriented Programming allows you to define a hierarchy of objects using *inheritance*. Inheritance promotes code reuse and semantic relations between objects. Subclasses provide specialization of behavior by allowing you to override object methods while preserving common functionality (generalization) defined in the super-class.

As a class-based object-oriented programming language, Java facilitates inheritance through subclassing by using the keyword extends. If class B `extend`s class A, B is said to be a subclass of A (A is a superclass of B). Instances of class B are also instances of class A, defining an "is-a" relation between them. Java also provides two related mechanisms related to subclassing.

- *Abstract Classes* – Java allows you to specify that a class is abstract using the keyword `abstract`. In an abstract class, you can define not only normal methods (which you provide a "default" implementation for) but you can also define abstract methods: methods that you do not need to provide an implementation for. Instead, it is the responsibility of non-abstract subclass(es) to provide an implementation. In addition, if a class is abstract, you are prevented from instantiating any instances

of it.

- *Interfaces* – Java allows you to define interfaces, which are essentially pure abstract classes. Interfaces specify the methods that a class must provide in order to implement the interface. Java allows you to define an interface that specifies the public interface (methods) of a class. Classes can then be defined to implement an interface using the `implements` keyword. One major advantage of interfaces is that it does not lock your classes into a rigid hierarchy; however objects that implement an interface can still be considered to have the is-a relationship. In addition, interfaces can be used to simulate multiple-inheritance in Java as classes can implement more than one interface.

Activities

You will explore these concepts by completing a Java program that simulates a basic weekly payroll reporting system for the Cinco Corporation. Every employee has an employee ID, a name (first and last), and a title. Further, there are two types of employees:

- Salaried employees – Salaried employees have a base annual salary, which is subject to a 20% income tax rate (state, federal and FICA combined). In addition, each salaried employee receives a $100 post-tax benefits allowance.
- Hourly employees – Hourly employees have a per-hour pay rate along with a weekly total of the number of hours they worked. Hourly employees do not receive any benefit allowance. Further, there are two types of hourly employees.
  - Staff employees are directly employed by Cinco and are subject to a 15% income tax rate.
  - Temporary employees are not directly employed by Cinco, but instead are contracted through a third-party temp agency who is responsible for collecting taxes (thus no taxes are taken from their gross pay).

Employee data is stored in a flat data file and the basic parsing has been provided. However, you will need to design and implement Java classes to support and model this payroll system. By the end of this lab your class designs should resemble something like that in Figure **??**.

Class Design & Inheritance

You have been provided a partially completed `PayrollReport` program that parses a data file (in `data/employee.dat`) and creates instances of an `Employee`. However, the `Employee` class is empty. You will need to implement this class and any relevant

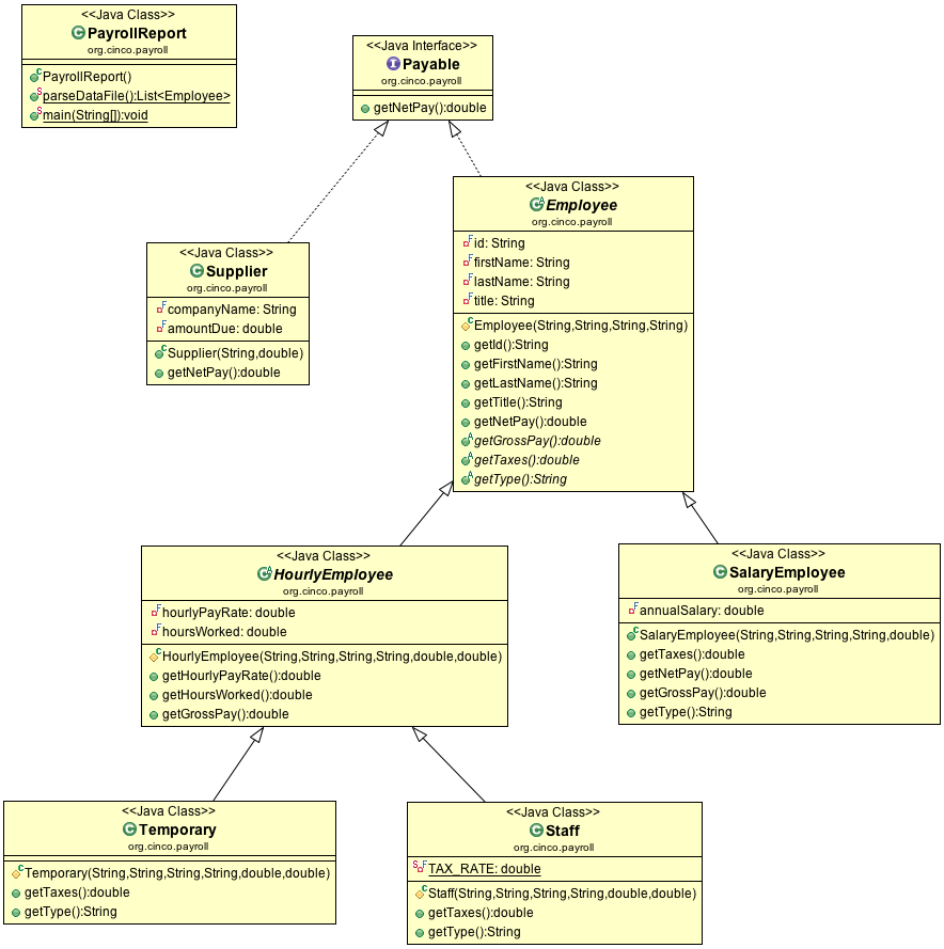

Figure 1: UML Diagram of a potential design for the Cinco Corporation payroll system

subclass(es) to complete the program.

- Identify the different classes necessary to model each of the different types of employees in the problem statement.
- Identify the relationship between these classes.
- Identify the state (variables) that are common to these classes and the state that distinguishes them. Where do each of these pieces of data belong?
- Identify the behavior (methods) that are common to each of these classes–what are methods that should be in the superclass? How should subclasses provide specialized behavior?
- Some state/behavior may be common to several classes–is there an opportunity to define an intermediate class?
- If you need more guidance, consult the UML diagram below on one possible design.

- To check your work, a text file containing the expected output has been provided in the project (see `output/expectedOutput.txt`)

Eclipse Tip: Many of the common programming tasks when dealing with objects can be automated by your IDE. For example: once you have designed the state (variables) of your class, you can automatically generate the boilerplate getters, setters, and constructors: when focused on your class, click "Source" → "Generate Getters and Setters" or "Generate Constructors using Fields."

Java Note: In a subclass, you *must* invoke a constructor in the superclass and it must be done first. This rule is so that classes conform to the is-a relationship. Invoking a super class's constructor is achieved by using the `super` keyword. If there are multiple constructors, you may invoke any one of them. Note that if a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the superclass does not have a no-argument constructor, you will get a compile-time error. The Java `Object` class has a no-arg constructor, so if your class does not explicitly extend a class, it implicitly `extends Object` and the no-arg constructor is invoked.

Abstractions

Now that you have a well designed, functional implementation we will improve on the design by identifying potential abstractions that can be made. In particular, start by making the `Employee` class abstract. If you had a good design, then nothing should break; the model did not have any generic employee–all employees were of a specific type. If something did break in your code, rethink your design and make the appropriate changes.

Further, identify one or more methods in the `Employee` class that could be made abstract. That is, are there any methods in the superclass where it would not be appropriate to have a "default" definition? Make these methods abstract and again make any appropriate changes to your design as necessary.

Adding an Interface

Cinco Corporation also has suppliers that they purchase supplies and parts from to build their products. Suppliers have a company name and an amount due, and so need to be paid, but they are not employees. However, the payroll department would like to have a common interface across all objects in their system that are, in some way, payable.

1. Create an interface (In Eclipse: right click the package → new → Interface) called `Payable`. Identify a single method in this interface that returns the (net) amount payable

2. Make the `Employee` class implement this new interface and make the appropriate changes if necessary
3. Create a new class to represent suppliers and also make it `Payable`
4. Answer the questions in your worksheet

Submission

We have included a test suite of unit tests written in JUnit (https://junit.org/junit5/) a popular unit testing framework for Java. Even though the test driver (in the `src/test` source folder) has no main method, you can still run it in Eclipse and get a report on how many of the tests passed, failed or resulted in an unexpected exception. Be sure all of the unit tests pass before submitting your source files through webhandin. You can rerun this test suite in the webgrader to ensure everything works.

Advanced Activities (Optional)

1. The `PayrollReport` class uses an `ArrayList` to hold instances of `Employee` objects. When it generates the report, it does so in the order that the instances were parsed from the data file. Change this so that the payroll report prints in order of the total net pay in decreasing order.
2. Unified Modeling Language (UML) is a common tool in Software Engineering that provides a visualization of the relationships between software components (subsystems, components, classes, workflows, use cases, etc.). Sometimes design of systems is done in UML and then tools can automatically generate Java (or other language) code conforming to the design. Conversely, UML diagrams (like that in Figure **??**) can be automatically generated from an existing code base using various tools. In this exercise you will familiarize yourself with UML and use such a tool to generate a UML diagram for your design.
    - Read the following tutorial on using UML for class diagrams: http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/
    - Install an Eclipse plugin for UML and generate a UML diagram for your project. The choice is yours, but one (free) possibility is ObjectAid UML:
        - Installation instructions: http://www.objectaid.com/installation
        - Generate class diagram instructions: http://www.objectaid.com/class-diagram