CSCE 156 – Lab: Inheritance

Handout

0. Prior to the Laboratory

1. Review the laboratory handout.
2. Read the following tutorial on Inheritance:
   http://download.oracle.com/javase/tutorial/java/IandI/subclasses.html
3. Read Abstract Methods and Classes and tutorial:
   http://download.oracle.com/javase/tutorial/java/IandI/abstract.html
4. Read Interface tutorial:
   http://download.oracle.com/javase/tutorial/java/IandI/createinterface.html

1. Lab Objectives & Topics

Upon completion of this 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

2. Problem Statement

Background

Object Oriented Programming allows you to define a hierarchy of objects through *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.

Since Java is a class-based OOP language, it facilitates inheritance through sub-classing by using the keyword `extends`. If class B *extends* class A, B is said to be a subclass of A. Instances of class B are also instances of class A, defining an "*is-a*" relation between them.

Java also provides two related keywords to achieve related OOP principles.

- 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 the "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 subclasses to provide an implementation. In addition, if a class is abstract, you are prevented from instantiating any instances of it.

- Interfaces – Java does not allow classes to extend (or inherit) from more than one class (some languages do allow *multiple-inheritance*). However, it is often useful to do so—to define a class that has multiple *is-a* relations. 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 (classes can implement multiple interfaces). Interfaces provide all of the advantages of multiple-inheritance without locking a class into a particular hierarchy.

Program

You will explore these concepts by completing a Java program that simulates a basic payroll reporting system. The University of China, IL (UCI) has two employee types, Faculty and Staff. Further, Faculty employees have three different subtypes, Assistant, Associate and Full Professor, each with a different annual salary. Staff employees are either full-time (salaried) or part-time (paid $8.50 hourly). The salary amounts for each employee are below.

Employee Type	Position	Annual Net Pay
Faculty	Assistant-Professor	$75,000
	Associate-Professor	$84,000
	Professor	$93,000
Staff	Full-Time	$41,000
	Part-Time	$8.50 / hour

The basic functionality for the payroll system has already been provided for you. The Payroll class loads employee data from a plaintext flat file and generates an end-of-year report for the annual net pay for each employee as well as their FICA (Federal Insurance Contributions Act) contributions. FICA is computed as 6.20% plus 1.45% of the total annual net pay for Social Security and Medicare respectively.

You will need to complete the program by defining several classes and implementing some of their methods.

Importing Your Project

A Java project has been started to implement a basic payroll system. Import this project into your Eclipse IDE by doing the following.

1. Download the zip file archive file from Blackboard (to your desktop or Z: drive)
2. Go to File -> Import -> General -> Existing Projects into Workspace
3. Select “Select archive file” and select the zip file you downloaded

Note: subsequent labs will provide similar Java projects. Refer back to this lab if for these instructions if you need to.

Activity 1: Interfacing

The publicly available interface of the Employee class is not sufficient to generate an annual report. In this activity, you will fix this by defining a couple of interfaces, make the Employee class implement them and provide implementations for the interface methods.

Instructions

1. Create two interfaces and have the Employee class implement them:
   - Payable – this interface should define one method that returns a double representing the total annual net pay of the employee.
   - Person – this interface should define two methods that return a string representing the first name and last name of the employee respectively. In addition, consider defining a method that returns a formatted name in a comma separated last-name, first-name order.
2. Implement the required methods in the Employee class and use them appropriately (hint: the `getAnnualFica` method should be updated and the new methods should be used in the Payroll's `printAnnualReport` method).

Activity 2: Sub-classing

Implementing the logic in one method to calculate the total annual net pay is complicated. Imagine a more complex class with many more cases and it quickly becomes unmaintainable and not easily extendable. It is also based on state that is not necessarily common to all instances (average hours per week). As an alternative, we will instead sub-class the Employee class and provide specialization by overriding methods.

Instructions

1. Create two subclasses of employee and use them appropriately in the Payroll program:
   - Faculty – a subclass of Employee used to represent faculty employees
   - Staff – a subclass of Employee used to represent staff employees.
2. Design these classes to provide specialized behavior by overriding methods. As a hint, keep in mind:
   - What is the common functionality that should remain in the Employee class?
   - What is the specialized *state* of each subclass?
   - What is the specialized behavior of each subclass?
3. Modify the Payroll's `loadFile` method as follows: instead of instantiating an Employee instance, it should either instantiate a Faculty or a Staff instance and add it to the employee collection depending on whether or not the data represents a Faculty or Staff employee.

Activity 3: Cleaning Up

Observe that, if you have designed and used your sub-classes well, you no longer have need to instantiate any instances of the Employee class. Moreover, the "default" behavior of the method that returns the total annual net pay has no meaning in the Employee class. This class is now a good candidate to be made abstract to avoid potential problems.

Instructions

1. Make the Employee class abstract.
2. Make the method that returns the total annual net pay abstract and remove any “default” implementation that you may have defined.
3. Handle any consequences of your code changes.

Demonstrate your design and working program to a lab instructor and answer the questions in your lab handout.

Advanced Activity (Optional)

1. Note that both the Faculty and the Staff classes could be further sub classed to provide even more specialized behavior. Think about how you would go about doing this. What methods would need to be overridden? What methods would be common to both? Implement your design.
2. The Payroll class uses a traditional array to hold instances of Employees. To do this, it instantiates an array that can hold at most 100 employees. This has many disadvantages, not only are we limiting the number of employees, but whatever fixed number we use, not all of them are necessarily used; we have to keep track of how many there are, and any other book keeping involved. A much better alternative is to use a List ADT that handles all of this for us by abstracting away the details of dynamically handling a collection of objects. The Java SDK (Standard Developer Kit) provides functionality for doing this through its `java.util.List` interface and its most common implementations, `java.util.ArrayList` and `java.util.LinkedList`. Modify the Payroll code to utilize these classes instead of using an array.