Indian Institute of Information Technology, Allahabad
Object Oriented Methodology (OOM) -2015
Lab Assignment-05

TAs: Bharat Singh, Nidhi Kushwaha, Balasaheb, Diksha, Daisy Monika, Yogesh Kumar, Saurabh, Ravi shankar, Rakesh, Sunakshi, Fahim Altaf

Date of assignment: 2/09/2015 **Due Date: 09/09/2015 (for Sec-A) & 10/09/2015 (for Sec-B)** **Instructor: Prof. O. P. Vyas & Dr. Ranjana Vyas**

Note: 1) All Assignments should be done independently.
2) Assignment should be evaluated within the deadline is essential.
3) Write simple code in java for printing appropriate value for class instances.

Q1. Write a java program to solve the following problem: A video rental store wants a program to keep track of its movies. It rents VHS and DVD movies, with each movie given a unique inventory number. Each customer must have a phone number, which is used as his or her membership number. The program needs to keep track of every customer and every movie, including information such as whether a movie is rented or available, who has it rented, and when it is due back. Employees of the store receive a commission on sales of non movie items such as candy and popcorn, so this information needs to be maintained as well. Determine the objects in the problem domain. For each object, determine its attributes and behaviors.

1. Write a Java class for each of the classes you came up. When writing your classes, keep the following in mind: Each class you write should be public; therefore, each class needs to appear in a separate source code file. Within each method, use the **System.out.println()** method to display the name of the method.

2. Writing a program to tie the classes together.

3. Write a program named **VideoStore** that creates an instance of each of your classes, initializes their fields, and invokes the methods.

Q2. Write a program that creates Elevator objects and invokes the various methods.

1. Write a class named Elevator. Add fields for the following attributes: an int for the current floor, an int for the floor that the elevator is heading to, a boolean to denote whether the elevator is going up or down once it reaches its destination, and a boolean to denote whether the elevator doors are open or closed.

2. Add a method named goToFloor() that changes the floor that the elevator is heading to. Use the System.out.println() method to display a message that you are changing the value.

3. Add methods named openDoors() and closeDoors() that change the appropriate boolean field accordingly. Again, display a message within each method so that you can see when the methods are invoked.

4. Add methods named goingUp() and goingDown() that change the appropriate boolean field accordingly.

5. Write a class named ElevatorProgram that contains main(). Within main(), instantiate two Elevator objects. Invoke the various methods of the Elevator class on these two objects, ensuring that all your Elevator methods work successfully.

Q3. The interface can contain any number of public methods. A class **Reactangle** implements the interface. Class Reactangle having field width, height of type double, and constructor, and methods **getperimeter()** and **getarea()**. There is another class **circle** having a field radius, constructor, and method **getcircumference()** in addition to method declared in the interface measurable. Define an interface measurable.java for methods that return their perimeters and areas by using getperimeter() and getarea() respectively. Create a display method in main which takes object of type interface measurable as argument.

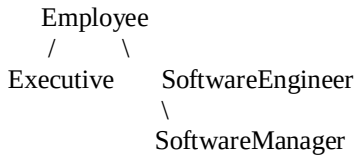
Q4. Write a program to help elementary school students practice simple addition and subtraction problems. The program should ask the user for the number of problems they would like to practice. When this value is entered, your program will randomly generate that number of addition and subtraction problems. The type of each problem generated will be random as well.

Implementation Details: Implement abstract class Problem that contains two integers representing operand s of a math problem. It will have two abstract methods:
• int getAnswer() – will return the answer of this math problem
• String getOperator() – will return a string representation of the operator for the problem

In class Problem, provide accessor and mutator methods for each of the operands. Then, implement two classes Addition and Subtraction that both inherit from Problem. In each of the subclasses, provide a c oncrete implementation of the super class abstract methods. Implement a main driver that follows the gene ral guidelines from the overview section. Your main driver should have at least three static methods in add ition to the main method. It should track the number of correct and incorrect answers given during the cou rse of game play. These counts should be displayed to the user when they are done with their problem set. The main driver should allow the userto start a new problem set or exit the program upon completion of o ne practice session. Be sure to include proper error checking where appropriate.

Q5. The question will involve relatively little code writing. The main goal of this question is to experiment with inheritance and understand how inherited methods fit into your code?

This question is built around a simple discrete event simulator. Specifically, we are simulating the day to day work environment of a software development company. Suppose this company has three types of employees: executives, software engineers, and software managers (i.e. the managers of the engineers). We can represent this organization structure with the following Java class hierarchy:



You will need the following starter files(see the attachments in this mail):

- Employee.java
- Manager.java
- CompanySimulator.java

The Employee Abstract Class

Here, we have an *abstract* Employee that contains code common to all of the company's employees. For example, all employees have a name, ID number, and the capacity to do work. Note that **Employee.work()** method is declared *abstract*. This method is only a stub and does not have an implementation. (all methods listed in an interface are implicitly abstract).

Because the Employee class is declared as abstract, Java will not allow you to instantiate Employee (example: new Employee()). Instead, we must define *concrete* sub-classes. A class is concrete if it is not abstract.

The backdrop code, CompanySimulator, is a simple discrete event simulator. At time *t*, the simulator runs through an ArrayList that represents the company and tells everyone to do work by invoking Employee.work().

Questions

1. Is it valid to define an abstract class that only contains fully implemented (i.e. not abstract) methods. If so, will Java let you create instances of this type? Try it.
2. Justify why the work() method is abstract?

SoftwareEngineer

The SoftwareEngineer class inherits from Employee as noted by the extends keyword in the class declaration. Software engineers implement the **work** method in the following way: engineers happily program, but cause a crisis 10% of the time. For our purposes, "programming" is defined as a simple "I am programming" print statement that identifies the engineer (using toString). You can generate a crisis by returning false. Otherwise, return true. Use the following code snippet to determine if a crisis should be generated:

```
Random crisisGen = new Random();
if (crisisGen.nextInt(10) == 0) {
   //it's a crisis!
} else {
   //everything is OK!
}
```

nextInt generates a random integer in the range [0, n-1]. In the above example, n=10. Crises are detected by the simulation code and triggers a call to **CompanySimulator.manageCrisis(Employee emp)** that you will implement. We will discuss the method further shortly.

Tasks

- Create the class **SoftwareEngineer** and inherit from **Employee**.
- Implement the **work()** method with an appropriate print statement: who this object represents and "I am programming".
- Implement the two SoftwareEngineer constructors: **SoftwareEngineer()** and **SoftwareEngineer(String empName, int empId)**. Make sure to call their respective super class constructor.
- Implement **toString**. Prefix the result of the Employee's toString with "SoftwareEngineer ".

SoftwareManager and the Manager Interface

The SoftwareManager is a specialization of SoftwareEngineer. Because the manager inherits from the engineer class, it is also an Employee. Note that in this example, our organization hierachy (managers are in charge of engineers), is the opposite of the class hierarchy. This is because the manager can perform engineering tasks (programming) and distinct management tasks.

In this lab, management tasks are defined by the Manager interface. Manager defines a single method: **handleCrisis**. The SoftwareManager not only inherits from a parent class, but also conforms to the Manager interface. In Java, classes are only allowed to inherit from a single parent class. However, classes may implement an arbitrary number of interfaces to simulate the multiple inheritance features of other languages while mitigating some of the problems that often arise.

Tasks

- Create the class **SoftwareManager**. You should inherit from **SoftwareEngineer** and implement the **Manager** interface.
- Implement **work()** and **handleCrisis()** with appropriate print statements: who this object represents and "I am programming" or "handling a crisis". How many methods do you need to write?
- Implement the two SoftwareManager constructors: **SoftwareManager()** and **SoftwareManager(String empName, int empId)**. Make sure to call their respective super class constructor.
- Implement **toString**. Prefix the result of the Employee's toString with "SoftwareManager ". Can you leverage the SoftwareEngineer.toString()?

Executive

Like the SoftwareManager, the Executive class inherits from Employee and implements the Manager interface. Executives also perform work (i.e. playing golf) and can handle crises. Overall, there is nothing particularly special about this class other than the fact that it is a higher position in the organizational hierarchy.

Tasks

- Create the **Executive** class. You should inherit from **Employee** and implement the **Manager** interface.

- Implement **work()** and **handleCrisis()** with appropriate print statements: who this object represents and "playing golf" or "handling a crisis".
- Implement the two Executive constructors: **Executive()** and **Executive(String empName, int empId)**. Make sure to call their respective super class constructors.
- Implement **toString**. Prefix the result of the Employee's toString with "Executive ".

Handling a Crisis

Recall that the **work()** method will return false if an employee has created a crisis. If a software engineer causes a crisis, the crisis must be dealt with by their respective manager. Determining the correct manager is dependent upon our chosen representation of the company's employees. More specifically, we use an ArrayList<Employee> to store all employees. This ArrayList is organized as follows:

Executive, SoftwareManager, SoftwareEngineer, SoftwareEngineer, ..., SoftwareManager, SoftwareEngineer, SoftwareEngineer, ...

An executive immediately preceeds one or more teams of software managers and software engineers. Exactly one manager preceeds their team of engineers. Thus, given a software engineer that causes a crises, you must locate the preceeding manager and invoke **handleCrisis()**. However, recall that software managers are simply specialized software engineers. Thus, software managers may also cause crises when working. In this case, you must find an executive to handle a software manager's crisis.

You may identify the employee types at runtime using Java's instanceof:

<object instance> instanceof <class type>

For example:

```
Mammal d = new Dog();
if (d instanceof Dog)
   System.out.println("It's a dog!");
else
   System.out.println("It's not a dog");
```

Note: Only the work method may throw a Crisis (as noted in Employee). An executive's work cannot cause a crisis.

Tasks

- Implement **CompanySimulator.manageCrisis(Employee emp)**. You may wish to test your implementation by modifying CompanySimulator.initCompany().

Q6. To see the working of polymorphism write a code with the below diagram, the use an object of type Liquid, you must create a Liquid object with new and store the returned reference in a variable:

```
Liquid myFavoriteBeverages = new Liquid();
```

The `myFavoriteBeverage` variable holds a reference to a `Liquid` object. This is a sensible arrangement; however, there is another possibility brought to you courtesy of polymorphism. Because of polymorphism,

you can assign a reference to any object that *is-a* `Liquid` to a variable of type `Liquid`. So, assuming the inheritance hierarchy shown in Figure 8-1, either of the following assignments will also work:

```
Liquid myFavoriteBeverage = new Coffee();
// or...
Liquid myFavoriteBeverage = new Milk();
```

Therefore, you can sprinkle some polymorphism in your Java program simply by using a variable with a base type to hold a reference to an object of a derived type.