Lab session #6 - Object Oriented Programming

Overview

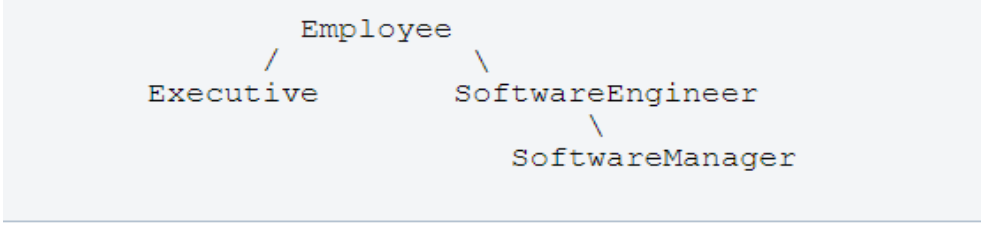
This lab will cover inheritance and polymorphism. More specifically:

- Inheritance
- Polymorphism
- `instanceof`
- Abstract Classes
- Multiple Inheritance via Interfaces

Lab Assignment

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

This lab 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:



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.

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 Array 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 1:

- 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 hierarchy (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 2:

- 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 "handling a crisis". 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 3:

- 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 "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 Array `Employee[]` to store all employees. This Array is organized as follows:

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

An executive immediately precedes one or more teams of software managers and software engineers. Exactly one manager precedes their team of engineers. Thus, given a software engineer that causes a crises, you must locate the preceding 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 4:

- Implement method **CompanySimulator.manageCrisis(Employee emp)**.

What to submit:

Your submission should include the following:

1. Lab report to answer all above questions.
2. Source code + README (how to compile and run your code).
3. Please create a folder called "yourname_StudentID_Lab6" that includes all the required files and generate a zip file called "yourname_StudentID_Lab6.zip".

Please submit your work (.zip) to Blackboard.