Problems relating to Unit 3: abstract and concrete classes, and polymorphism

1. Consider the following inheritance hierarchy diagram:

Vehicle

Car

Tank

Assuming you have already written the Vehicle, Car and Tank classes (if you haven’t, write stub classes for them with headers that correspond to the hierarchy), write an application which declares and creates an ArrayList called fleet of Vehicles and adds one car and one tank to it. Explain how polymorphism can be used to display the details of each vehicle in the fleet. If we need to write a Train subclass of Vehicle, and add a Train to the fleet, will we need to change the generic part of the application that displays the subclass instances in a polymorphic manner? Explain your answer.

1. Explain what you understand by the OO term **polymorphism**.
2. Explain very briefly what is meant by the term **dynamic method binding** – illustrate your answer with references to the code from the previous exercise.
3. What is the fundamental difference between an **abstract** and a **concrete** class? Give *any* suitable example you like of each type.
4. Where do **abstract** classes normally appear in an inheritance hierarchy? Where do **concrete** classes normally appear in an inheritance hierarchy?

The sample classes Employee, Manager, HourlyWorker and WorkerTest:

1. Copy the files Employee.java, Manager.java, HourlyWorker.java and WorkerTest.java to your H: drive. First examine the **Employee.java** file and answer the following questions based on it.
   1. The class is defined to be abstract, which, among other things, means that the class should not be instantiable. In the driver program WorkerTest.java, add code that will attempt to instantiate an Employee object. What is the outcome? Why does it make sense that the Employee class should be abstract?
   2. Try to make the Employee class a regular concrete class, by removing the word abstract from its class definition header. Recompile. What happens? What does this tell you about abstract methods?
   3. Notice that the Employee class does not contain any no-argument constructor and yet the application compiles fine. How is this possible?
   4. The Employee class contains just one abstract method called earnings(). Would it be possible to create an abstract class without any abstract methods in it? Try it out and recompile. What is the outcome?
   5. Notice that the abstract method is written without any opening and closing curly braces after the (). Would you expect the program to compile if braces {}were put in before the semi-colon? Try it out and recompile. What is the outcome? What if you omit the semicolon? Must the class still be declared abstract?
2. Examine the Manager.java file and answer the following questions based on it:
   1. How can you tell that Manager is a concrete class?
   2. The Manager class is defined to be final. What does this mean?
   3. Does Manager actually inherit the attributes defined in the Employee class? Prove this.
   4. Explain very briefly what the purpose of the following line of code is in the Manager constructor: super(fn,ln);
   5. Looking at the code for setWeeklySalary(), what would you expect to happen if a value such as –1000 was input for the Manager’s weekly salary?
   6. The Manager class contains an earnings() method. Does this method absolutely have to be in the class or could it be omitted or called something else e.g. managerEarnings()? Try out these possibilities and report your results. What does this tell you about the inheritance of abstract methods?
   7. The toString() method in the Manager class makes a call to the Employee toString() method in order to do its job, via the ‘super’ reference. Is it essential that the method toString() be called on the super reference or could it be omitted?
   8. Give any example within the Manager class where overriding occurs.
3. Examine the driver program WorkerTest.java and answer the following questions based on it.

(a) The driver program here contains polymorphic behaviour. Explain where the **polymorphism** occurs.

(b) Indicate exactly where in the driver program **dynamic method binding** occurs.

(c) Notice in the driver program the lines of code such as

eRef=b1; and

eRef=hw1;

where the superclass object reference is set to refer to appropriate subclass objects. Do you think it should be possible to do the opposite i.e. have the subclass objects set to refer to the superclass reference? Try it out to see the result. Does this make sense?