Sw Development Inheritance exercise 4

Part 1

Implement a superclass Person. Create two additional classes, Student and Instructor, that inherit from Person.

A person has a name (string) and an age (int).

A student additionally has a degree (string) and an instructor has a salary (double).

Write the class definitions, the constructors, accessor methods and the toString() method for all classes. Use the supplied test program, Exercise4Tester1, to verify your classes.

Note: It would be good practice to write the methods yourself, but you can use IntelliJ's code generation facilities, if you wish. Please refer to Appendix A for information on generating toString() for subclasses.

Check list:

- Make all instance fields private
- Use super() to chain the constructors together. Make sure that the order of parameters in your constructors matches with the tester
- The toString() method in each subclass *could* utilise the toString() method in the superclass to help it perform its task (hint: we looked at using `super.method()` recently). Try to implement it like this[1].

```
public class Exercise4Tester1{
   public static void main(String[] args){
      Person p1 = new Person("Mike Smith", 24);

      Student s1 = new Student("Mary Jones", 19, "Applied Computing");

      Instructor i1 = new Instructor("John Jacobs", 55, 41500.70);

      //Print out each object's details via toString()
      System.out.println("Person details : " + p1.toString());
      System.out.println("Student details : " + s1.toString());
      System.out.println("Instructor details : " + i1.toString());

      //Print out just the student's name, for example
      System.out.println("Student's name is : " + s1.getName());

      //Print out just the instructor's salary, for example
      System.out.println("Instructor Salary is : " + i1.getSalary());
```

[1] Dfsdf

```
    }
}
```

Part 2

Following on from part 1, create a new tester class (call it Exercise4Tester2).

NOTE: because we will be using an ArrayList in this exercise, try to avoid having named objects (e.g. p1, s1, and i1 in the previous part)

1. In your main method declare an empty ArrayList of Person objects - call it `people`, for example.
2. Add a Person object (reference) to the ArrayList . Hint: use the `new` operator to create a new Person object reference.
3. Add a Student object to the ArrayList. This should work for you if you do it correctly (i.e. in the same manner as you added the Person object previously).
   What we've just done is important: our ArrayList stores Person references but the reality is that ***the reference could actually be a subclass of Person***, as is the case here.
4. Add an Instructor object to the ArrayList.
5. Use a for loop (enhanced or traditional) to iterate over the ArrayList and print out each object's details using `toString()`.
   If you've done it correctly, you'll notice that the **correct** toString() is automatically invoked for each object. This is actually a very important outcome and it demonstrates a property in object-oriented programming know as polymorphism.

Part 3

We saw in part 2 that we could add subclass references (Student, Instructor) to an ArrayList which we declared would hold Person references. **Because we provided overridden versions of the toString() method, the correct version for each object was invoked at run-time.**

Now, create a new tester class (call it Exercise4Tester3)

1. Again, in your main method declare an empty ArrayList of Person objects – call it `people`, for example.
2. Add **one person object and two student objects** to the ArrayList – again, this should be fine with the compiler.
3. Use a loop to iterate over the list printing out **each person and student's name**. Make sure it works.
4. Now, attempt to use another loop to iterate over the list printing out each student's degree. You should notice that this won't work – any ideas why? Well, because our ArrayList was of type Person, we can only invoke methods that are actually declared in the Person class (and, of course, getDegree() is not). We can get round this issue, and we'll look at how to do that soon.

   For those of you who want to investigate straight away, one solution involves the use of the `instanceof` *operator and then casting appropriately.*

Appendix A - Using IntelliJ to generate toString() for subclasses

As you hopefully know, you can generate a number of useful methods (including constructors) from IntelliJ – Click on the "Code" and select "Generate" or just use the shortcut keys Alt+Insert (which means hold down the Alt key and then press the Insert key).

If you select the option to generate `toString()` for a subclass, you can select one of several templates. The screenshot below shows a useful one for subclass generation.