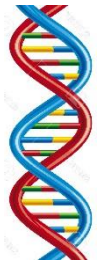
Programming Exercise 14:

Inheritance and Polymorphism



Purpose: Gain experience in extending a base class and overriding some of its methods.
Background readings from textbook: Liang, Sections 11.1-11.5.
Due date for section 001: Monday, April 25 by 10 am
Due date for section 002: Wednesday, April 27 by 10 am

---

1. Overview of Inheritance

We've already seen the utility of defining our own classes. So far, we've examined two important aspects of OOP: encapsulation (defining data and methods together in a class) and information hiding (the ability to make some aspects of a class private). Two other aspects of OOP are *inheritance* and *polymorphism*, which we explore in this lab. Inheritance is the ability to take a class and extend it to a new class. This new class has added functionality. The original class is known as the *base* class or *parent* class and the extended class is known as the *subclass* or *child* class. The child class is more specific by containing extra instance data and more specialized methods. Let's consider the Student class we introduced several labs ago. We might decide that a GradStudent will be treated similarly but have some differences. For instance, a normal student may have little research potential but a GradStudent is expected to do research. Or, there may be different ways to compute scholarships for Student versus GradStudent. The GradStudent class will inherit all instance data and methods from Student but might add additional instance data like degreeEarned (String) and workExperience (boolean) and additional methods like computeManagerPotential. Additionally, a method defined for Student, say computeScholarshipOpportunities might be revised in the GradStudent class. We might also extend Student into a second child, LawStudent which will have its own instance data and methods. A LawStudent may not have any scholarship opportunities so we could redefine in LawStudent the computeScholarshipOpportunities method from Student to always return 0.

In order to use inheritance, we have to first make a change to our parent class. In order for something defined in a parent class to be inheritable (whether instance data or methods), it must be visible and so cannot be private, yet we want to make all instance data hidden from outside classes. So we use a different visibility modifier called `protected`. Protected items are not visible from outside the class except for subclasses. Anything that is public will remain public.

Once our parent class is written and compiled, we can write our child class. To denote that this is a child of a specific parent, we add `extends ParentClassName` to the class header as in

```
public class GradStudent extends Student
```

The GradStudent class now has everything from Student that was public or protected. Now we can define GradStudent's specific instance data and methods. We would make the instance data protected if we believe that GradStudent itself might be a parent for other classes, and private if we do not intend to have a child class of GradStudent. We define any appropriate new methods while also redefining methods that we inherit from Student that will operate differently from Student. So for instance, we would probably include accessors and mutators of the new instance data. We do not need to redefine accessors and mutators for Student's instance data unless we will vary how they operate. We will also need to define constructors for GradStudent and possibly a new toString since

GradStudent has more instance data to return as part of the String. For these methods, we can actually reference Student's versions to save us some of the effort. Let's consider Student's 0 parameter constructor. It initialized all instance data to 0 or "unknown". Now for GradStudent we have two additional instance data. Rather than defining a constructor for GradStudent that initialized all of Student's instance data plus the two new instance data for GradStudent, we call upon Student's 0 parameter constructor and then initialize the two new instance data. To reference some of the parent class, we use the word `super` (super means "superclass", or parent). Here we see two constructors for GradStudent.

```
public GradStudent() {
      super();
      degreeEarned="unknown";
      workExperience=false;
}

public GradStudent(String n, String m, double g, int h, String d,
      boolean w)
{
      super(n, m, g, h);
      degreeEarned=d;
      workExperience=w;
}
```

The Student's toString method returned the relevant parts of the Student's instance data as a String. We want GradStudent to return the same information but also any extra information from the new instance data. Let's assume we want GradStudent's toString to also return the degree they had already earned. We can do this as follows.

```
public String toString() {
      return super.toString() + "\t" + degreeEarned;
}
```

2. Overview of Polymorphism

Polymorphism sounds like a genetics term but in fact it's not a very complex topic at all. Consider when you declare a variable in Java. From that point forward, that variable must always be of the specified type. For instance `int x` means x is an int and `String y` means y is a String. However, when it comes to declaring objects, Java gives you a little bit of flexibility. Consider the following two declarations:

```
Student s1;
GradStudent s2;
```

s1 is a Student and s2 is a GradStudent. Seems simple. But the way Java implements objects, s1 can also be a GradStudent. That is, we could do the following:

```
s1 = new Student(…);
s2 = new GradStudent(…);
…
S1 = new GradStudent(…); // even though s1 is a Student
or  s1 = s2;             // it can also be used to reference
                         // a GradStudent
```

The variable s2 cannot be set to a Student, but s1 can be set to a GradStudent. The ability to assign an object variable to be any subclass is polymorphism. Note that if you do not extend a class, it automatically inherits from the top-most Java class called Object. So interestingly, `Object foo;` would be allowed to take on any type of object that has been defined in Java.

This allows you to have flexibility in that once a variable is declared at compile time, it can still be assigned different types at run-time based on run-time circumstances such as input from the user. For instance, we do not know whether the user will want to create a Student or a GradStudent. We declare s1 to be of type Student. We input from the user which type of Student and using an if statement we do the following:

```
if(userInput.equals("student"))
     s2 = new Student(…);
else s2 = new GradStudent(…);
```

Let's assume that Student has a method called computeScholarshipOpportunities and that is has been redefined in GradStudent. An interesting situation arises when we have the instruction `s2.computeScholarshipOpportunities();`. Which method is actually invoked, that of Student or GradStudent? Without polymorphism, this decision is made at compile time. If we had GradStudent s3; then s3 could only be a GradStudent and s3.computeScholarshipOpportunities(); would invoke GradStudent's method. But with s2, the compiler cannot know what will happen at run-time and so the decision is not made until run-time. This concept is known as *dynamic binding*. The method call is bound to the actual method of a class at the time the method call is reached. This is less efficient than compile time binding.

Let's consider another situation. We have defined a method in GradStudent called potentialForResearch. This method is not defined in Student. Now consider the following code.

```
Student s1;
GradStudent s2;
…
s1.potentialForResarch(); // generates a syntax error
s2.potentialForResearch();  // does not
```

The first call to the method generates a syntax error because Student does not have this method. The interesting thing here is that s1 *could be* a GradStudent. But to play safe, Java generates a syntax error. To avoid this, you must cast s1 as a GradStudent by doing the following:

```
(GradStudent(s1)).potentialForResearch();
```

This additional cast tells the Java compiler that s1 will be a GradStudent when this code is reached so that this method can be invoked.

Part 3: Creating Your Base Class

In this assignment, you will create four classes, a base class, two child classes and a user class. The base class will be a BasketballPlayer class. The two child classes will be CollegeBasketballPlayer and ProBasketballPlayer. The BasketballPlayer class will have instance data of name, position, team (Strings), height, weight, agility, speed, ballHandling (ints) and all should be declared as protected instead of private. We will have 3 constructors, a 0 parameter constructor (all instance data are

initialized to "unknown" or 0), a 3-parameter constructor which receives the 3 Strings and assigns the 3 String instance data to them and 0 to all int instance data, and one which receives a parameter for each instance data and assigns them all appropriately. Also create accessors for all instance data and a toString which will return at least the name, position, team and the value for a method you will write called getValue. The getValue method will test various instance data and return an int using nested if-else logic based on the table below. All methods should be public.

Position	Other conditions	Value
Center	height >= 82, weight between 220 and 250, ball handling > 5	10
Center	height >= 80, weight between 210 and 260, ball handling > 5	8
Center	height >= 80, ball handling > 4	6
Center	height >= 78, agility > 7	5
Center	height >= 78	4
Center	height >= 76, agility > 5	2
Center		0
Forward	height >= 80 and either agility > 8 or speed > 7	10
Forward	height >=78, agility > 6, speed > 5	8
Forward	height >= 77, agility > 5	6
Forward	height >= 76, speed > 4	5
Forward	height >= 75 and either agility > 3 or speed > 3	3
Forward	height >= 74	1
Forward		0
Guard	height >= 78, ball handling > 7 and either agility or speed > 7	10
Guard	height >= 76, ball handling > 7 and either agility or speed > 6	8
Guard	height >= 74, ball handling > 5, agility > 5, speed > 6	6
Guard	ball handling > 6, agility > 6, speed > 5	4
Guard	ball handling > 4, agility > 4	2
Guard		0

NOTE: empty "other conditions" mean "any values", these would be used in else clauses.

Compile this class before continuing as you cannot implement the child classes without this one being compiled. Fix any syntax errors.

4. Creating Your Child Classes

Implement a CollegeBasketballPlayer to extend BasketballPlayer. It will have two instance data, eligibilityRemaining (int) and role (String). Note: these can be protected or private as we do not plan to extend this class. Have 3 different constructors, a 0 param constructor, a 3 param constructor (name, position, team) and a constructor that receives all params. Use `super();` for the 0 param constructor, `super(…);` for the other two constructors, passing the appropriate parameters. Assign eligibilityRemaining and role to either 4 and "bench" for the 0 and 3 parameter constructors or the proper params for the third constructor. Add accessors for these two instance data a new toString method which return `super.toString() + "\t" + "…"` where … is at least the role of this player (and the eligibilityRemaining if desired). Finally, add a method called draftable which returns a boolean based on the following logic:

> A player is draftable if he/she is a "starter" (role) whose value (as obtained by super.getValue()) > 4 or is a "bench" player (role) whose value >= 8.

Compile this class and fix any errors.

Implement ProBasketballPlayer by extending BasketballPlayer. Add 2 new instance data: yearsInLeague (int) and role (String). Add 3 constructors similar to what you had for CollegeBasketballPlayer. For the first two constructors, yearsInLeague should be initialized to 0 and role to "bench". Add accessors for the two new instance data and a toString which will add to BasketballPlayer's toString both yearsInLeague and role. Finally, implement a method called newContractValue which returns an int value based on the logic in the following table.

Value (using getValue())	Years in League	Role	Return value
	< 3		0
> 8	>= 10	Starter	12,000,000
> 7	>= 8	Starter	10,000,000
> 7	>= 5	Starter	8,000,000
> 5	>= 10	Starter	6,000,000
	>= 5	Starter	2,000,000
		Starter	1,000,000
> 8	>= 10	Bench	7,500,000
> 7	>= 8	Bench	5,000,000
> 5	>= 10	Bench	4,500,000
	>= 7	Bench	2,000,000
		Bench	500,000

Again, empty entries in the table mean "any value". NOTE: because of the logic here, you may need to add `else return 0;` at the end of your nested if-else statement to satisfy the Java compiler.

Compile this class and fix any errors.

5. Creating a User Class and Submitting Your Assignment

Finally, create a user class with an empty main method. On the website, next to the link for this assignment is a link for some Java code. Copy that into this user class to be the main method. In the code, you will see three variables, bp1, bp2 and bp3, declared as a BasketballPlayer, CollegeBasketballPlayer and ProBasketballPlayer respectively. Notice that bp1 could take on an object of BasketballPlayer, CollegeBasketballPlayer or ProBasketballPlayer but bp2 and bp3 are limited to just their declared type. The code then performs various instantiations and message passing on bp1, bp2 and bp3. You will notice some of the instructions have syntax errors. Before trying to compile the program, comment out the instructions that have syntax errors and add in comments next to those instructions the reason for the errors.

Examine the source code and add comments whenever a method is being invoked via dynamic type binding. Finally, compile and run your program and capture the output. Submit your four classes and your output. For convenience, you can copy all 4 classes and your output into a single text file or word document, or you can send all files separately. The output can either be in a file by itself or pasted to the bottom of your user class in comments.