Object-Oriented Programming (in Java)
Assignment 2: Creating Subclasses of Actor

Introduction
In your previous lab assignment, you created and tested an Actor class. In particular, you created two methods: set() and toString().
The Actor class had four instance fields which identified characteristics that will be common to all types of actors.
In this lab assignment, you will:
  • derive two subclasses from the Actor class, namely, the subclasses Hobbit and Wizard.
  • create an array of references to Actor, and store the reference-to values for several Hobbit and Wizard objects.

Modifying the Actor Class
Very little needs to change in the Actor class. That’s one of the goals of object-oriented programming: code re-use (that is, using existing code without having to modify it).
There will be one tiny (but essential) change to Actor. It must become an abstract class (that is, a class which will never be used to create actual Actor objects). The Actor class will be used as a super class for all the other types of actors. (We’re starting with Hobbit and Wizard, but you’ll see in later labs how easy it is to expand with dozens of other subclasses.)
When you add the keyword abstract to your Actor class specification, you may notice that your main() routine flags this kind of code as a syntax error:

```
Actor aActor1 = new Actor();
```

Error Message: Cannot instantiate the type Actor.

Good! It should be a syntax error now. Remember, an abstract class can never be used to create an object. From now on, we will create Hobbit objects and Wizard objects, but never again will we create an Actor object.

Inventing the Hobbit Class
Look to the lab exercise for the syntax to create subclasses (a subclass is a class that inherits from some super class). In that earlier exercise, you worked with classes Employee, Manager and Scientist. You can also look to Chapter 9 of your textbook.
In this assignment, we want to create a Hobbit subclass based on the Actor superclass. (Note: You’ll need to use the keyword extends.)

Very little needs to be added to the Hobbit subclass. That’s one of the goals of object-oriented programming: code re-use (without having to modify it).

Additional Hobbit Instance Field
In The Hobbit and Lord of the Rings, hobbits were small people who were hardly ever noticed. Naturally, we want to capture this ability in an instance field, so I’m suggesting the addition of the following:

  • nStealth: The value in nStealth captures of how easily a Hobbit can move without being seen.

Hobbit Virtual (Polymorphic) Methods
The Hobbit class will implement the set() and toString() methods. These will be virtual (polymorphic) methods (virtual because the superclass Actor defines implementations of set() and toString()).

The principle of code re-use suggests that we use all the functionality of the superclass Actor and merely add the new detailed requirements for Hobbit.
So, the set() method might look like this:

```
public void set() {
  super.set();
  nStealth = Input.referenceToSingleInputObject().getInt("Stealth: ", 0, MAX_SPEED);
}
```

In the preceding code the set() method in the Hobbit class immediately calls the set() method of the superclass (that is, the Actor class). This allows the superclass to do the work that it needs to, then the Hobbit class manages the things that are unique to it.
You’ll need to do something similar for the toString() method.
Note about toString(): The String object that’s built by toString() is best formatted as a single line of text, without extra newlines.

Inventing the Wizard Class
Now, we want to create a Wizard subclass based on the Actor superclass. Again, very little needs to be added to the Wizard subclass.

Additional Wizard Instance Fields
In The Hobbit and Lord of the Rings, Gandalf the Grey often carried a staff – and the presence of his staff infused him with extraordinary powers. Naturally, we want to capture this characteristic in an instance field, so I’m suggesting the addition of the following:

  • bHasStaff: The boolean value (true / false) in bHasStaff will be used at a later time in determining how the wizard can move. With a staff, teleporting might be possible.

In Lord of the Rings, Gandalf also had a horse called Shadowfax. With his horse, Gandalf could move at great speeds.

  • bHasHorse: The boolean value (true / false) in bHasHorse will be used at a later time in determining how the wizard can move.

Wizard Virtual (Polymorphic) Methods
The Wizard class will implement the set() and toString() methods in ways that are similar to the Hobbit class.

Array to Hold Actor SubClass Objects
There’s a very important design technique when using inheritance and virtual methods.
When creating and working with subclass objects, you must use a reference to the superclass to hold the location information of the subclass object. For example, in main() when testing your Hobbit and Wizard classes, you’ll do something like this:
Actor a1 = new Hobbit();
a1.set();

Actor a2 = new Wizard();
a2.set();

Actor a3 = new Wizard();
a3.set();

a2 = new Hobbit();
a2.set();

You might be wondering, “Why do we have to do this?” Here’s why!

We will ultimately be managing hundreds . . . perhaps thousands of Actor objects. We CANNOT manage them by single named variables. We MUST use a tool to manage them as a collection. To start, we’ll use an array. But what will be the data type of the array? It MUST be an array of references to Actor objects. Even though it will be an array of references to Actor objects, it can store the location information (the reference value) for any subclass type. So you could create the array like this:

Actor[] armyActors = new Actor[1000];

You could create an army of Hobbits and a couple of Wizards like this:

armyActors[0] = new Wizard();
armyActors[1] = new Wizard();
for (int i = 2; i < armyActors.length; ++i) {
    armyActors[i] = new Hobbit();
}

You now have a mix of two different types (Wizard and Hobbit) managed by using a single array. You can probably imagine how that single array could manage many different types: Hobbit, Wizard, Orc, Goblin, Human, Dwarf, etc. But that will come later.

Test Plan

In your previous lab work you created a separate class to test your Actor class. For this lab work, you’ll have to modify the testing class to confirm that your inheritance design works and that execution can bind to the correct virtual (polymorphic) methods.

Create and manipulate several Hobbit objects and several Wizard objects to prove that each object retains its separate identity, and to prove that execution correctly threads through the virtual method calls.

When you test the program, there are several categories of testing:

- **Boundary Condition**: Are there high and low boundaries for variables such as: nStrength, nSpeed, nHealthPoints. These boundary conditions should be handled by the set() method. Can you correctly validate the capture of boolean values.

- Do the virtual (polymorphic) method calls arrive at their correct locations? (This one is probably best tested using the debugger and using Step Into <F5> to inspect execution. Remember to check the Debug window (also known as the call stack) to see if you’ve stepped in too deeply. If so, you can step back out again with <F7>). Testing the execution of virtual methods involves checking both set() and toString().

### Building JavaDoc Documentation

Use JavaDoc comments in your code and then generate the browser-ready documentation from those JavaDoc comments. You can look to my Input class for examples of JavaDoc syntax. You can also look to the Oracle / Sun site for JavaDoc details:

www.oracle.com/technetwork/java/javase/documentation/index-137868.html

### Memory Map

Show the reference-to the array of references-to Actor, and how the subclass objects are managed using the array.

A Hobbit object will have five instance variables (the four inherited from Actor and the one specialized instance variable unique to the Hobbit, namely nStealth). A Wizard object will have 6 instance variables (the four inherited from Actor and the two specialized instance variables unique to the Wizard, namely bHasStaff and bHasHorse).

Drawing the memory map will convince me that you understand the difference between primitive variables and reference-to variables, and how an array of references-to objects (using the superclass) can manage actual subclass objects.
**Problem Statement (1 mark)**

**Memory Map (2 marks)**

**Test Plan (1 mark)**

**Program Code**
- Test Plan pre-condition, post-condition assertions match `assert` statements in program code (1 mark)
- Pattern of Indentation / Comments / Variable / Method / Class Naming (1 mark)
- Embedded JavaDoc comments and generated JavaDoc (1 mark)

**Program Runs Correctly**
- Generates correct output (1 mark)
- Data validation (1 mark)
- Meaningful labels and messages (1 mark)

**Subtotal:**  / 10

**Bonus (up to 3 additional marks):**
- Useful, clever, clear enhancements
- Good use of GUI
- Other

**Penalties (unlimited):**
- Redundant code (repeated code that could be eliminated or split into a separate method)
- Dead code (code that could never possibly be executed)
- Other

**Total:**  / 10

---

**Due Date:** By Friday February 15th at a time and in a manner to be determined by your lab professor. (Note: The following week is the break week. I will published Lab Assignment 3 before the break so that you'll have it during the break week.)

Your lab professor may publish further information about submitting the lab work.