COM1300 Lab #10

To complete this lab you must be familiar with Chapter8.   Chapter 9 is helpful.

The BlueJ project for this lab is hosted on a zip file in Angel. Copy the project to your personal "Z" drive, extract it, edit it, and keep your work for a few weeks at least. As before, make sure your Angel dropbox submission works! Test it by re-downloading your file under another name and opening it.

Objectives of this Lab:

- More practice with using inheritance on the Zuul project.
- Exploring method polymorphism
- Review of some previously learned programming skills

You are going to add more dynamism to the Zuul game in the form of roving monsters (OK, just one type of annoying professor-monster is required).   Start with the Zuul project from last week that you have already enhanced with portable items.   (You only need the item code working if you tackle an extra-credit challenge option). When you are finished, zip up your Zuul project and submit it to the drop box.

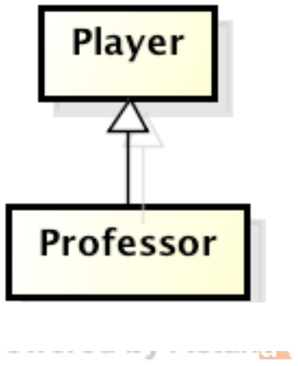
The end-goal is that a professor-monster starts out in some assigned room and starts moving randomly through Zuul, helpfully(??) pointing out directions. When the player enters a room where there is a professor, or just looks around in the current room, part of the room's long description will be a sentence: "There is a professor here; the professor is pointing <direction>." where <direction> refers to one of the possible directions north, south, east, or west. This sentence should only appear when there is a professor in the room. In reality, the direction does not indicate anything special. The professor just moves from room to room and points at random.

1. In Lab 6 you did an exercise to pick any one of a list of choices at random.   We will use that technique to give our new "monster" some behavior. Open your Lab 6 zip file and find the `throwDice` method you wrote in the `RandomTester` class. Based on what you have learned since then, make any necessary corrections to this method before proceeding. In particular, this method must *return* a number between 1 and 6, not just *print* it.

2. Open your Zuul project simultaneously with your Lab 5 project. Soon, you will need to copy your implementation of `throwDice` into one of the Zuul classes, but there is design work to be done first.

3. Up until now, there has only been one active "actor" in Zuul: the player. Last week you moved some player-related state information formerly in a single Game object into a Player class. This week we build on that.   Most of the capabilities of a player need to be shared by monsters as well. To keep things simple, we will make each monster class a direct subclass (extension) of the player class, so that the only distinction between a player and a monster is the extra A.I. that enables a monster to act autonomously.

Add a new monster class to Zuul as shown below.



4. Regression test your game at this point to make sure it still plays perfectly before you actually add professor-monster objects.

5. Make sure that the Professor constructor calls the Player constructor as described in class.

6. Add a copy of the throwDice method to the Professor class. For testing purposes you may leave this public, but make it private before you turn in your lab. This will require a few other additions to the Professor class:
   a. Importing java.util.Random
   b. Adding a Random object to the fields of the Professor class, to be used in throwDice.
   c. Making sure the Monster constructor properly initializes the Random object

7. The rules for professor motion are the following. Each professor has an act() method. Whenever this is called, the professor rolls a simulated die. If the die comes up 6, the professor will move from its current room to a new room, picking an exit at random. Implement this method using the throwDice method to determine whether or not to move, then use the Random object a second time to pick an exit. (Hint: Remember in Lab 5 how you programmed a Random object to pick one element at random out of an ArrayList.)

   To implement this step you will need to be able to generate a list (i.e., an ArrayList) of the exits in the current room. One way to do this is as follows:
   a. Add a new public method called getExitList to the Room class; this method takes no parameters and returns an ArrayList of strings.
   b. The body of this method can be adapted directly from getExitString, only instead of concatenating the exits into a long string, you will add them to a (local) ArrayList and then return the arraylist.

   You are not obligated to use this method, but it is one way to get a list of the exits from a room. Another way, but messier, is to use the Room class' getExitString more directly. If you make this method public, then you can use it to return a concatenation of all exit directions. Then, in the Professor class, you can split this string into an array of individual words, discard the first one ("Exits:") and stuff the rest of them into an ArrayList.

8. Each professor has an additional piece of state: the direction in which it is pointing. It only changes this direction when it enters a new room. Add a new String field `pointingDirection` to the Professor class to represent this. Extend the `act` method so that each time a professor enters a new room, this field must be set at random to one of the four directions north, south, east, or west. You can either do this in the `act` method itself or by defining a private method `point` called by the `act` method.

9. Add a method called toString() to the Professor class, taking no parameters and returning a String. This method should return a sentence telling which direction the professor is pointing, *e.g.*: "There is a professor here. The professor is pointing north." This will be printed out at the *end* of the room description whenever a professor is present.

10. Unit test your Professor class to make sure the act method works. Create a mini-game on the Object Bench by creating two Room objects, `room1` and `room2`. Set two exits in `room1` leading to `room2`. Create a Professor object and place it in `room1`. Call the professor's `toString` method and make sure that it is behaving properly.

    Next, call the `act` method repeatedly until the professor moves to room2 . (Since `room2` has no exits, the professor will stay in `room2` once he gets there.) This is a bit of a pain because there is no set limit on the number of times `act` most be called before the Professor moves. But if `throwDice` is programmed properly, he is almost certain to move within 20-30 tries.

11. Add a field to the Game class to hold a reference to a Professor. In the createRooms method, create a Professor in this field and assign it to one of the rooms.

12. In the Player class, create a **static** ArrayList of type Player, initialized statically to an empty ArrayList. Call this list **actors.** Every time any Player (or Monster) gets created (the Player constructor), make sure that the object gets added to that list

13. There are multiple places in the Game class where the game prints a long description of the current room. Change these so that the Professor's `toString` method is also printed out after all the other information, but **only** if the player and the professor are in the same room! (Think how you can use the **actors** list to simplify this task.)

    Test this code by playing a game and sending the player into the same room as the professor. The professor's information should be printed out when the player is in the same room, and at no other time.

14. Change the game's `processCommand` method so that the professor's `act` method is called after each command the player executes. Thus, every command the player executes risks that the professor will "get distracted" and wander off. Play a game and make sure

that the professor is wandering occasionally. Also test that the professor doesn't change what he's pointing at until he moves to a different room.

Challenge Problems

There are two challenge options for this week's lab. Both require a bit of thought. I suggest you try only one of them unless you have plenty of time. If you try both, you will get full credit (10 points) for the one with the best score, and a maximum of 2-3 extra points for the other one.

15. [Challenge #1] OK, so professor-monsters that just point at exits are kind of boring. What other kinds of simple actions should monsters be able to do? Add a Thief monster as another direct subtype of Player. When its act() method is called, it rolls a die. If the die comes up 5 or 6, a Thief picks up one object in the room and adds it to his bag. If the die comes up 1, he takes an object at random out of his bag and drops it in the current room. If the die comes up 3, he just moves at random to a different room through one of the exits. Test your thief similarly to the way you tested your Professor before you introduce him to an actual game. Make sure he works OK in a room with no objects or when carrying an empty bag.

16. [Challenge #2] A game with no objective is pretty lame, so let's add one. Add a new portable item to your game called the "Amulet of Wisdom" (apologies to Zork). Place the amulet in a random room when the game starts up. The objective of the game is to get the amulet, find a professor, and drop the amulet in the room with the professor. When you do that, you graduate (win the game)!! Add this logic to your game, including a congratulatory message to the winning player. (Note that it is not sufficient for the Amulet to simply be in the same room with the Professor; the player must actually drop it there to win the game.)

Jar up your BlueJ Zuul project and upload to the dropbox. <u>Verify that your upload worked.</u>

Honors Problem (2 weeks to complete this)

One of the essential skills mentioned in Chapter 6 was modifying a program with low risk by re-factoring it. In this exercise, you will use refactoring to change the underlying representation for the location of items in Zuul, without changing how the game works In this exercise you will be creating a *alternative* version of you project that should not replace your main copy, so make a copy of your project before you attempt this refactoring and keep it separate. Use the Honors Dropbox for your alternative project. You will still need the main copy of your Zuul project one more time for the last lab of the semester.

The goal of this exercise is simple. You are going to change the way the location of items is recorded in the game *without* disabling the existing *public* interfaces for any of the classes. You may be tempted to add one or two more public methods, but try not to. You should do as much as you can using private code changes. Don't expose any more than you have to publicly. If you feel a real need to extend the public interface of a class, please discuss it with me first via email!

This is how items are to behave in your alternative game:

- Each item has a new private field `location` which holds a reference to a `Room`. That room is the current location of the item.
- The `location` field of an item is `null` when it currently being carried by the player.
- The player's `bag` collection and each room's `items` collection should no longer be used and should be removed by the end of the exercise.
- All existing public methods should continue to do that same thing as they do now, but they may have to be re-written internally in order to do so. Do not change the name, parameter list, or return values for any existing public function.
- Hint: to make this work, you will find that you need to create and keep a master list of all items currently in the game. I suggest that you create this as a private static ArrayList or HashSet in the Item class. That way, when an item is consumed (cookie, etc.) you can delete it from the game by removing it from the master list..

Re-factoring means that you will have the new code running alongside the old code before you start to disable and remove the old code. At no time in this exercise should your code be so torn up that it can't compile and run.

Certain optional features of your game may be hard or impossible to duplicate in this new version. For example, the thief may not work because there is only provision for one player who can carry items. You may disable the thief code from your project if you want to in order to complete this assignment. This is the reason for making this version an alternative Zuul and not your main version.

Jar up your *alternative* Zuul project and upload to the Lab 10 Honors dropbox. Verify that your upload worked.