Subclasses

Purpose:
The purpose of this lab is to review specifications and to learn to implement simple subclasses.

Setup:
Create a directory named Lab25/mazegame in your Java directory. Copy all the .java files from ~labCourse/Labs/Lab25/mazegame to this directory.

The MazeGame:
The classes considered in this lab model rooms and players in a simple “adventure” game. Rooms are connected with passages. A room can have four passages leading in the directions north, south, east, and west. We assume that the passages don’t bend, so that if moving north from room A puts you into room B, then proceeding south from room B put you back into room A. However, we assume no more about the layout of the rooms. For instance, as illustrated below, moving north and then east from a given room will not necessarily get you to the same place as moving east and then north.

A player can be in a room, and for simplicity, a room can contain only one player.

MazeGame Documentation:
Documentation for classes Lab25.mazeGame.Player and Lab25.mazeGame.Room can be found here. A class Lab25.mazeGame.Maze is also described, but this is just a utility class at present.

- The specification for Maze calls NORTH, SOUTH, EAST and WEST “variables.” They are not variables, but named constants. Does it matter that the documentation doesn’t tell you what their values are?
- What does the Room query room tell you? Suggest a better name for this method.
- What does the Room query isPassage tell you? Notice that this method takes an integer as an argument. The argument must be one of Maze.NORTH, Maze.SOUTH, Maze.EAST, Maze.WEST.

**Testing the classes:**

The file TestIt.java contains a skeletal “test driver” for the Room and Player classes. Edit the test driver and make sure you understand it. Modify it so that you perform more complete testing of the Player and Room classes.

**The classes Room and Player:**

Look at the source for the classes Room and Player.

- The Room methods setOccupant, etc. are listed as “restricted.” The documentation indicates that they should only be invoked by certain objects under specific circumstances. Why have these “restrictions”? Why not just let these methods be ordinary public methods?

Recall that a dynamic interaction diagram illustrates how objects interact as clients and servers to accomplish some task:

- Draw an interaction diagram illustrating the scenario of a Player moving from one Room to another. (You diagram should show three objects: the Player and the two Room’s.)

- Look at the specifications for the methods setOccupant and deleteOccupant in the class Room. Look at the method move in the class Player. Are the preconditions for setOccupant and
deleteOccupant guaranteed to be met? What global assumptions about the game do you need to guarantee that the preconditions for setOccupant are met?

- Modify the specification of move to conform to the requirements of setOccupant.

Extending classes:

We will now define a new kind of Player called a Wizard. A Wizard has an additional move method, in which the target Room is specified rather than the direction:

```java
public void move (Room newLocation)
```

We also define a new kind of Room, called an EnchantedRoom. This kind of room has no additional properties. However, using the above method, a Wizard can move directly to an EnchantedRoom. That is, for the above move method to work, the specified Room must either be directly connected to the Wizard’s current location, or must be an EnchantedRoom.

There are a few items to note about subclasses.

- Constructors are not inherited; constructors should be explicitly defined in a subclass.

- The constructor of a subclass explicitly or implicitly calls a constructor of its superclass as its first action. The keyword super is used to call a superclass constructor. For instance, the constructor for the Wizard class should look something like this:

```java
public Wizard (String name) {
    super(name);
    // possibly other stuff
}
```

If we didn’t explicitly call the Player constructor with super(name), a call super() would be automatically inserted. This would cause an error, since Player does not have a constructor with no arguments.

- A subclass does not “inherit” private components from its parent class. For instance, if we look at the code for the Player, we notice that a Player has a private name component. Since a Wizard is-a Player, a Wizard instance will have such a component variable. But since the variable is labeled private, it is not “inherited” and cannot be directly accessed from within the definition of the class Wizard. That is, the definition of the class Wizard could not contain a statement like

```java
this.name = “FooBar”;
```

that directly accesses the variable. (It could, however, contain an invocation of its changeName method.)

- The boolean operator instanceof can be used to determine if an object is an instance of a given class. For example, if p is a Player variable, we can determine whether the object which p currently references is in particular a Wizard, as follows:

```java
if (p instanceof Wizard) ...
```
(We will learn later that excessive use of the `instanceof` operator is often an indication of faulty design.)

Now do the following:

- Add classes `Wizard` and `EnchantedRoom` as described above.
- Compile and test. (Of course, you will need to modify `TestIt` somewhat.)

**Post-lab:**

Submit the following, as directed by your lab instructor:

- listings of your classes `Wizard` and `EnchantedRoom`, and of your modified `TestIt`;
- your modification of the `Player` method `move`;
- a transcript of your test.