Object-Oriented Programming (in Java)
Lab Assignment 4: Moving Actors

Introduction
We have worked through the key elements that are to be part of your final submission. These include:

- **JavaFX framework** implemented with skeleton menu and map of Middle Earth.
- **Ability to populate** two Army objects with a selection of Actor subtypes.
- **Ability to display** Actor avatars as Node objects in a JavaFX scene.
- **Ability to display** Actor attributes on the console screen (that is, ordinary text output through System.out.println() or System.out.printf()).
- **Ability to display** Army of Actor objects in a ListView.
- **Ability to display** Army of Actor objects in a TableView, accessing only the Actor superclass attributes (none of the subclass attributes). Ability to edit an Actor object’s attributes. Changed attributes will be visible in the console display.
- **Ability to start and suspend** the movement of Actor objects.

The Required Elements

There are two elements to bring this project to closure:

- Complete the subclass rules for moving Actor objects.
- Revise UML diagrams.

There are two optional elements that some of you may embrace, but they are purely optional (with associated bonus marks):

- When an Actor object moves, if it is close enough to an opponent, engage the moving Actor and the opponent in combat. This will adjust the health of an Actor. If the Actor object is too weak to move, it will stop moving. If the health drops below zero, the Actor will die (which presents some interesting challenges about multi-threading and thread-synchronization).  
- Add a menu item and implement the code to save the current Battlefield state to disk using serialization. Add a related menu item to restore the two Army objects from disk.

Details: Moving Actor Objects
Consider a case where you are dealing the movement calculations for a single Actor. There are two layers to define movement:

**Superclass: Actor**
The `startMotion()` method already exists in the Actor class. Each time it is called, a single Actor object defines the next leg of its journey—a process that is defined by the TranslateTransition attributes. Here are things to consider:

- The Actor’s health could be used to define a delay in starting motion (using `tt.setDelay()`). Very low health could indicate that the Actor is wounded. But be careful about the mathematics: health is a reciprocal of duration, that is, a smaller value for health means a larger delay. (So, you’ll have to divide health by some suitable constant to determine a delay in seconds).

- Each Actor has a speed attribute (which is accessed using `getSpeed()`). The value associated with speed must be converted to a value representing the duration of motion (in seconds). That double value for speed can be used in the call to `tt.setDuration()`. But again, be careful about the mathematics: speed is a reciprocal of duration (so, you’ll have to divide speed by some suitable constant to determine a duration in seconds).

**Subclasses: Hobbit, Wizard, Orc, Elf . . .**
The `startMotion()` method now finds the nearest opponent (by asking the opposing army to find the nearest Actor object). You should already have this code from the lecture build, but let’s review how this works. I’ve captured the essential code here, and had added commentary below.

```java
/** Defines the characteristics of a <i>TranslateTransition</i>.
* Each call results in ONE segment of motion. When that segment is finished, it “chains” another call
* to &lt;i&gt;startMotion();&lt;/i&gt; (which is NOT recursive!)
* The initial call is made by the managing &lt;i&gt;Army&lt;/i&gt; object; subsequent calls are made through
* the “chaining” process described here.
*/

public void startMotion() {
    Army opposingArmy = army.getOpposingArmy(); // NOT RECURSION!!!
    System.out.println("ToMove:[%.1f:%.1f] Opponent:[%.1f:%.1f]"
                   .format(getAvatar().getTranslateX(),
                           getAvatar().getTranslateY(),
                           opposingArmy.getOpponent().getAvatar().getTranslateX(),
                           opposingArmy.getOpponent().getAvatar().getTranslateY()));

    // You must implement the findNewLocation() methods for each subclass
    // The newly created Point2D object will contain the newly calculated X,Y values
    Point2D newLocation = findNewLocation(opponent); // NOT YET FULLY IMPLEMENT. This is YOUR work.
    ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    if ((tt.getStatus()) == Animation.Status.RUNNING) {
        // TODO: Implement to use newLocation
        tt.setTranslateX(newLocation.getX());
        tt.setTranslateY(newLocation.getY());
        tt.setDuration(Duration.seconds(1.0));
        tt.play(); // give assembled object to the render engine.
        // Of course, play() is an object-oriented method which has access to "this" inside,
        // and it can use "this" to give to the render engine.
    } else if (RUNNING) {
        // end if (RUNNING)
    } else if (RUNNING) {
        // end startMotion()
    }
}
```

Review of Existing and Completed Code

- The `startMotion()` method is an Actor method, called by the single Actor object that is trying to determine where to move.
- The Actor object knows its own Army allegiance, that is, the Actor object has a reference-to the Army that holds it in its ArrayList&lt;Actor&gt;.
- Each Army object now knows about the opposing Army. So, the startMotion() can ask my Army for the opposing Army, then ask the opposing Army to findNearestOpponent(). By sending this as an argument to the `findNearestOpponent()` method, the opposing Army has access to my location information (stored in my avatar object’s TranslateX and TranslateY properties).
- The `findNearestOpponent()` method returns a reference-to the nearest opposing Actor.
- The call to `findNewLocation()`, will bind to the correct subclass implementation. If the subclass is a Hobbit, then the new location will be away from the opponent. If the subclass is a
Wizard, then the new location will be toward the opponent. The Wizard’s distance will be influenced by whether the Wizard has a horse or his a staff.

- The `findNewLocation()` method will create a new `Point2D` object to hold the X and Y values for the new location. Creating a `Point2D` object is important. In Java, we can only return one thing from a method. If we tried to return primitive `double` values for the X and Y, we have a problem: there are two values. We can create a single `Point2D` object that contains both the X and Y values. We can then return that single `Point2D` object with its two values.

- You can see that I’ve modified the line of code that resets the `TranslateTransition` X and Y values. It now uses the `Point2D` object (which we can access through the reference-to variable named `newLocation`).

Bottom line: There is very little code to change in this method. Nearly all your work is involved in setting up the subclass methods for `findNewLocation()`.

**Summary of your Task**

The in-class lab work resulted in Actor objects moving randomly on the screen. You are to implement an algorithm where each Actor object looks for its nearest opponent then moves in relation the nearest opponent. The nature of the movement will be managed at the subclass level, for example, Hobbits move differently than Orcs (Hobbits run away, Orcs run towards).

The movement of each Actor object is driven by a `TranslateTransition`. The controlling attributes for the `TranslateTransition` should be adjusted by a particular Actor’s attributes. An Actor whose speed is greater will will have a shorter duration for the `TranslateTransition` (in other words, the `TranslateTransition` finish sooner). An Actor’s health, could be used to influence the delay in starting the `TranslateTransition`. Other subclass attributes can also be factored into the `TranslateTransition` attributes.

**Details: UML Diagram**

I have demonstrated the use of Visio during labs and lectures. I have shown the elements that should be part of the UML diagram that will accompany your submission.

**Optional Bonus: Managing Battles**

**Using the `combatRound()` Method**

This is NOT a requirement, but most of you already have the `combatRound()` method. You could modify the `startMotion()` method to see if the Actor being moved is close enough to its nearest opponent to engage in combat. If close enough, let the two Actor objects engage in a skirmish.

**Removing Dead Actor Objects from an Army**

If the health of an Actor drops to zero, that Actor is dead and can be removed from the Army’s collection of Actors. This can be quite challenging because of the multi-threaded nature of this application. This is quite an advanced topic which you will explore formally in semester 4 (you’re only currently finishing semester 2). If any of you are up to the challenge, I am ready to help you.

**Submission**

Your lab professor will provide further details about your final submission, but it will include the following. Printable material should be assembled in a single electronic document.

Here’s the list of elements that will be evaluated in your submission.:

- **Problem Statement**: What is the purpose of each of the move methods that you are implementing?
- **Test Plan**: The test plan need only identify the elements being tested for this phase, that is moving Actor objects.
- **Program Code**:
  - JavaDoc documentation.
- **UML diagram**: Your diagrams will follow the style described during lectures and in your textbook.

**Submission Due Date**:

You have the rest of the semester to complete this work. However, the precise due date and time will be defined by your lab professor.