ESS 201 Programming II (Java)
Java Lab 6
Mini Project
v2

The objectives of this assignment are:

- build on the object-oriented design principles discussed in this course
- demonstrate how abstract classes and interfaces enable multiple members of a team to work independently and still collaborate on a project.
- get an exposure to threads and concurrency

Each of you should implement the program below. For the final evaluation, teams of 6 students will integrate their classes (as described later) and demonstrate how these implementations interact in an integrated setting.

This exercise sets up an animation of geometric shapes such as would be useful for a simulation, game or other graphics applications. In this simulation, we have a set of stationary objects (obstacles) and a set of moving objects (actors). Actor objects move around autonomously from a start position to a destination position. In the process, they compute a path such that they will not intersect with obstacles or other actors.

The animation involves recomputing the position of each actor at each time step, and displaying the objects in their current position.

For collision detection, we use orthogonally aligned rectangular bounding boxes, the smallest rectangle that contains the points of the object. These bounding boxes (BBox) are defined by their corners with the min and max coordinates. Two objects collide if their BBox'es touch or overlap.

Hence, for animation, each actor computes its next position (given a timestep and its destination position). If this new position will not collide with any other object, it moves to that position. If not, it computes a different (non-colliding) position and moves there. If no such position exists, it stays in its current position until it can move. You are free to implement any collision avoidance scheme, so long as the actor moves towards the target destination. Note that the destination location of the actor can be changed anytime (by main, for instance).

We model this with the following classes/interfaces:
class **Scene**: holds the set of obstacles and actors, drives the animation, and updates the View. Extend this class to implement missing functionality (especially *checkScene*)
class **View**: mechanism for showing the position of the objects. A default TextView is provided. A graphical view class will be provided
interface **BBox**: definition of bounding box and interface methods. Provide a concrete implementation of this.

class **Point**: a helper class to define a point in 2D
class **SceneObject**: the main class that will need to be extended. More details below.
Each student should define a package containing the derived classes of Scene, BBox, SceneObject, as well as any other helper classes. For consistency, the package should be the roll number: imt20xxyyy
The main will import the relevant packages, and instantiate the appropriate derived objects

SceneObject extensions should support the following:

1. creating an object. You can choose any shape you would like, defined by a set of boundary points. The size of shape should not exceed 40 units in x or y
2. update position given a time step. This involves checking for intersections and arriving at the next position. To keep the animation consistent, the movement in any timestep should not exceed 20 units in x or y.
3. compute the bounding box. Return the derived class of BBox that you have implemented
4. To enable the display, you have 2 options:
    a. use an icon/image. Return the name of the file (bmp/jpg) that contains an image you would want to use. This will be rendered at the position where the object is.
    b. Return the list of points that represents the outline of the object. This will be used by View if needed. Note, this should be implemented even if the image file is provided.

Note: in the implementation of checkScene in class Scene, you should detect if any of the actors is violating collision conditions. If so, you can take appropriate actions, such as removing them from the list of actors (effectively removing them from the simulation).

Please see the uploaded files:

- Directory/package animation contains the base classes/interfaces. You should **not** change any of the content in this directory.
- directory/package demo: you will need to create a similar package (with the rollno as the name) and implement the derived classes here
- Demo.java: will be replaced by the actual test file. Also, a new View implementation, DisplayView, will be provided.

For the demo/evaluation:

- Each of you will use your implementation of main, Scene and other default classes, and add in the packages/implementations of the rest of the team. We run the demo on your machine using these classes
- This is repeated for each of the team members.

Notes (Added in v2):

1. The coordinate system for the Scene, both for display and animation, has the origin (0,0) at the lower-left corner of the rectangular region. x increases to the right, and y to the top.
2. Instances of each derived class of SceneObject are created using the default constructor of that class. You can decide if multiple calls to the constructor return the objects with the same data or different data. For example, you could store some static data that determines the list of points and/or images used by each instance of the derived class. You should assume that your derived class constructor will be called at least 2 times, as obstacle and as actor. The main will explicitly set their initial position after constructing them. Note that destination is set only for Actors and not for Obstacles.
3. You can keep the logic for collision avoidance simple. It may be necessary for your object to back-track for a short while to avoid obstacles. What is needed is that over the course of the demo, your actor should move largely in the direction of the destination
4. We can assume that obstacles are not too close to each other. Your algorithm for deciding on the path need not be very complicated, and can use a local/greedy approach. As an example, on detecting potential collision, the object could choose to temporarily change direction to the left or right. Note that if an object is to move by (dx,dy), then (-dy,dx) and (dy,-dx) provide directions that turn left or right.
5. Your derived class of Scene should check for validity at each time step (checkScene), and should identify any collisions. The action you take on collision is up to the implementation. For instance, you could visually indicate where the collision is, or, even consider removing the colliding actor from subsequent animation steps.
6. class Scene has an additional static method to get the last created instance of the Scene class. This could be useful for the SceneObject to query for the set of other sceneObject instances.