CS 051 Homework Laboratory # 4
Boxball

Objective: To gain experience defining constructor and method parameters and using active objects.

—

The Scenario.

For this lab, we would like you to implement a game called Boxball. This is a simple game in which the player attempts to drop a ball into a box. Boxball has 3 levels of difficulty. With each increasing level, the box gets smaller, and the player is required to drop the ball from a greater height.

When the game begins, the playing area is displayed along with three buttons that allow the player to select a level of difficulty. Selecting a level of difficulty affects the size of the box and the position of the line indicating the minimum height from which the ball must be dropped. The player drops a ball by clicking the mouse in the region of the playing area above the minimum height line. If the ball falls completely within the box, the box is moved to a random location. Otherwise, it remains where it is. In either case, the player may go again, dropping another ball.

The boxball playing area should appear as follows:

boxballpic

The online version of this handout has a demo version of BoxBall. See the online version of this handout for instructions on obtaining a copy of the starter folder and for links to see the starter code.

Design of the program.

For this lab you will need to define three classes: a `Box` class, a `Ball` class, and a `Boxball` class that extends `WindowController`. These will allow you to create the major components of the game, i.e., the box, the balls, and the playing area, respectively. You are expected to come to lab with a design for each class that corresponds to the functionality specified in parts 1, 2, and 3 described below. 15% of your lab grade will depend on the design you bring to class. Even if you further refine your design in lab, you will be graded on the design you bring to lab.

To give you a better sense of what is expected for a design, we provide you with the design for `MagnetGame` and `Magnet` classes from last week's lab at the end of this handout.

The `Boxball` class should be responsible for drawing the playing area when the program starts. It should also handle the player's mouse clicks. If the player clicks on an easy, medium, or hard button, the starting line should move

to the appropriate height. If the player clicks within the playing area above the starting line, a new ball should be created and dropped from that height.

The `Box` class will allow you to create and manipulate the box at the bottom of the playing area. A box is responsible for knowing how to move to a new random location when the ball lands in the box. It also is responsible for changing size if the player changes the difficulty level.

The `Ball` class should allow you to create a ball that falls at a constant rate. When the ball reaches the bottom of the playing area, the player should be told whether the ball landed in the box. The ball should then disappear. If the ball lands in the box, the box should move to a new location.

Part 1: Setting up Start by setting the layout of your playing area, using the familiar `objectdraw` shapes. Our playing area has both width and height of 400. The code for building the playing area should go in `Boxball.java`. (The window itself should be set to be 600 by 600 when creating a new run configuration in Eclipse.)

Once you have set up the playing area, add the Easy, Medium, and Hard buttons to your layout. The buttons are just rectangles that will respond to mouse clicks. After you've displayed the three buttons, add code to the `onMouseClick` method that will adjust the level of the starting line, depending on the button clicked. If the player selects the "Easy" buttton, the line should be relatively low. If the player selects the "Hard" button, the line should be quite high.

Part 2: Adding the box Next, you should add a box to your layout. To do this, you will need to write the `Box` class that will allow you to create and display the box object at the bottom of the playing area.

A box is really just a rectangle, but there is some important information you will need to pass to the `Box` constructor in order to construct it properly. First, you need to tell the box where it should appear on the display. Remember that its horizontal position will change over time, but its vertical position will always be the same. What are the extreme left and right values for its horizontal position?

In order for the rectangle to be drawn, you will also need to tell the box what canvas it should be drawn on. This information will be passed on to the constructor for the rectangle.

The box needs one more piece of information for it to be drawn correctly. It needs to know how wide it should be. Of course, even in its hard setting, the box should still be a little wider than the ball. Since the `Boxball` class will create both the ball and the box, it knows their relative sizes. It must therefore tell the box how big it should be when the box is created.

Once you have written the constructor for the `Box` class, you should go back to the `begin` method of the `Boxball` controller class and create a new `Box`.

The default setting for the game is "Easy". If the player clicks on the "Medium" or "Hard" button, the box should get smaller (or much smaller). The box needs a method, `setSize`, to allow its size to change when the player

clicks on Easy, Medium, or Hard. Think carefully about what parameters you need to pass to `setSize` to accomplish this command.

After writing the `setSize` method, test it by modifying the `onMouseClick` method in the `Boxball` controller. Clicking on one of the level selection buttons should now not only raise or lower the bar, but it should also adjust the size of the box.

Part 3: Dropping a ball The `Ball` class is different from other classes with which we have been working. Objects of this class will be active.

The player will create a ball by clicking with the mouse in the playing area above the starting line. When the click is detected, a new `Ball` should be constructed. The constructor for the `Ball` class should draw a ball at the appropriate location on the screen. It should also start it moving down the screen. To do so, it will call a `start` method, which will cause the code in `run` to execute.

You will notice that the skeleton of the `Ball` constructor we have provided for you contains the call to `start`. You will need to add more statements to the constructor, but you should be sure that the `start` method call is the last statement in your constructor.

You need to think carefully about what information a ball needs to know to construct itself properly. Recall that `Boxball` knows how big the ball should be (so that the ball and the box can be sized appropriately). `Boxball` also knows where the mouse was clicked. This should be the starting location for the ball. You need to pass this information to the `Ball` constructor so that it can draw itself and fall.

To get the ball to fall, you will need to do some work. This will be done in the `run` method. We have provided a skeletal `run` method for you.

We will use a bit of video game magic to make the ball appear to fall. The ball will appear to move smoothly down the screen, but in fact, it will be implemented by a series of movements. Each movement should move the ball a short distance, wait a short time, and then move again. The `run` method contains a while loop to allow this. On each iteration of the while loop, the ball should move a small distance, say 10 units. Then it should wait a short time. The pause method call that we provided in the starter tells the ball to wait 50 milliseconds. There are 1000 milliseconds in a second. Moving short distances that rapidly will appear to be continuous movement to the human eye. This is the same technique that television and movies use to provide continuous motion. To complete the while statement, you need to provide the condition that determines when to exit the while loop. Specifically, the ball should stop moving when it reaches the bottom of the playing area.

Once you have written the `Ball` constructor and the `run` method, you should test them. Return to your `Boxball` controller class, and add code to the `onMouseClick` method that will construct a ball if the player clicks above the starting line in the playing area. At this point, do not worry about whether the ball falls in the box. Just check that it is drawn at the right starting location

and that it makes its way to the bottom of the playing area.

Part 4: Checking the box Now you are ready to determine whether the ball fell in the box. As the ball reaches the bottom of the playing area, it should compare its location to the box's location. Of course, the ball will need to find out the box's location. The `Box` class needs to provide methods `getLeft` and `getRight` that give the positions of the edges of the box. To call those methods, the `Ball` class must know about the box. Go back and modify your `Ball` constructor to pass in the `Box` as an additional parameter.

If the ball lands in the box, you should display the message "You got it in!". If the ball misses, you should display "Try again!". Since the `Boxball` controller is responsible for the layout of the game, it should construct a `Text` object that displays a greeting message. The `Text` object should be passed to the `Ball` constructor as a parameter so that the ball can change the message appropriately when it hits or misses the box.

Test the additions that take care of checking whether the ball fell in the box.

Finally, use the random number generator (i.e., `RandomIntGenerator`) to pick a new location for the box when the player gets the ball in the box. The box should be responsible for picking the new location and moving itself. Therefore, you will need to add a method to the `Box` class called `moveBox`.

When the box is narrow it's left can edge can have a relatively large value while still having the whole box in the playing area. However, when the box is wide, it cannot move quite as far without having its right edge going outside the playing area. You may set up the random number generator to only give values that will result in the widest box staying inside the playing area. For extra credit, you can further refine the program so that even the narrower boxes can end up all the way on the right of the playing area.

Due Dates

You get a bit more time this week, so this assignment will be due at 4 pm Wednesday afternoon.

Submitting Your Work

Before submitting your work, make sure that each of the .java files includes a comment containing your name. Also, before turning in your work, be sure to double check both its logical organization and your style of presentation. Make your code as clear as possible and include appropriate comments describing major sections of code and declarations. Use the Format command in the Source menu to make sure your indentation is consistent.

Turn in your project the same as in past weeks. Use the Export command from the File menu. Check that the Boxball project is being exported to a new folder whose name includes your name and identifies the lab. Then quit Eclipse and drag your folder to the dropoff folder.

Table 1: Grading Guidelines

Value	Feature
	Design preparation (3 pts total)
1 pt.	Boxball class
1 pt.	Box class
1 pt.	Ball class
	Style (7 pts total)
2 pts.	Descriptive comments
2 pts.	Good names
2 pts.	Good use of constants
1 pt.	Appropriate formatting
	Program Quality (5 pts total)
1 pt.	Good use of boolean expressions
1 pt.	Not doing more work than necessary
1 pt.	Using most appropriate methods
1 pt.	Good use of if and while statements
1 pt.	Good choice of parameters
	Correctness (5 pts total)
1 pt.	Drawing the game correctly at startup
1 pt.	Changing box size and line height correctly
1 pt.	Dropping the ball
1 pt.	Determining if the ball landed in the box
1 pt.	Moving the box after the ball lands in it

Also, before turning in your work, be sure to double check both its logical organization and your style of presentation. Make your code as clear as possible and include appropriate comments describing major sections of code and declarations. Refer to the lab style sheet for more information about style.

MagnetGame Design

```
/*
 * Name:  Joe Cool
 * Lab:   Lab 3 (magnets)
 * Purpose:  MagnetGame draws two magnets on the screen.  It handles
 *           mouse input to allow the user to drag and flip the
 *           magnets.
 */
public class MagnetGame extends WindowController{

    /* The two magnets */
    private Magnet magnet1, magnet2;

    /* These variables are used to remember which magnet the user
       is actively moving and which is reacting to the magnetic field
       of the moving magnet. */
    private Magnet restingMagnet, movingMagnet;

    /* Remember where the mouse was on the last press or drag so
       we know how far to move the magnet. */
    private Location lastPoint;

    /* Remember if we are dragging a magnet. */
    private boolean dragging = false;

    // Draws the magnets on the screen
    public void begin()  {
        // Draw the two magnets
    }

    // Remembers the mouse position and if the user is pointing at
    // one of the magnets.
    // Parameters:  point - where the user pressed the mouse button
    public void onMousePress(Location point) {
        // Remember where the mouse was pressed in lastpoint.
        // If the first magnet  was pressed on,
        //      remember that magnet as the movingMagnet and the other
        //      magnet as the restingMagnet.  Set dragging to true.
        // else if the second magnet was pressed on
```

```
        //      set it as the movingMagnet and the other as the restingMagnet.
        //      set dragging to true.
        // otherwise (we are in neither magnet),
        //      set dragging to false.
    }

    // If a magnet was selected, drag that magnet.
    // Parameters:  point - the current mouse location
    public void onMouseDrag(Location point)  {
        // If dragging is true,
        //      determine how far the mouse moved
        //      and move the movingMagnet by that amount.  Then check
        //      to see if that movement causes an interaction with the
        //      resting magnet.
    }

    // If a magnet is clicked on, flip that magnet.
    // Parameters:  point - where the user clicked the mouse
    public void onMouseClick( Location point)  {
        // If the first magnet was clicked on,
        //    flip that magnet and the see if the flip causes an
        //    interaction with the other magnet.
        // else if the second magnet was clicked on
        //    flip that magnet and the see if the flip causes an
        //    interaction with the other magnet.
    }
}
```

Magnet Design

```
/*
 * Name:  Joe Cool
 * Lab:   Lab 2 (magnets)
 * Purpose:  Magnet models a physical magnet with a north and south
 *           pole.  A magnet can move and it can interact with other
 *           magnets through attraction and repulsion.
 */
public class Magnet {
    /* The dimensions of the magnet. */
    private final static double MAGNET_WIDTH = 150;
    private final static double MAGNET_HEIGHT = 50;

    /* Distance from pole to magnet perimeter */
    private static final double POLE_DISTANCE = MAGNET_HEIGHT / 2;
```

```
/* The rectangle representing the outline of the magnet */
private FramedRect box;

/* The poles of the magnet */
private Pole northPole, southPole;

// Creates a new magnet
// Parameters:
//    point - the upper left corner of the magnet
//    canvas - the canvas that the magnet will be displayed on
public Magnet (Location point, DrawingCanvas canvas) {
    // Draw the rectangle
    // Create the north and south poles offset by POLE_DISTANCE
    // within the rectangle
}

// Return the upper left corner of the magnet.
public Location getLocation() {
    // return the location of the box;
}

// Move the magnet relative to its current position.
// Parameters:
//    xoff - the x offset in the movement
//    yoff - the y offset in the movement
public void move(double xoff, double yoff) {
    // Move the rectangle and the two poles by the same offset.
}

// Move the magnet so that its upper left corner is at the given
// point.
// Parameters:
//   point - the new upper left corner for the magnet
public void moveTo( Location point) {
    // Calculate an offset from the point to the current box
    // location.  Call the move method using the calculated
    // offset.
}

// Returns true if point is within the bounds of the magnet.
// Parameters:
//   point - the point to check for containment
public boolean contains (Location point) {
    // Return whether the box contains point.
}
```

```
    // Swaps the north and south poles of the magnet.
    public void flip ()  {
        // Remember the x coordinate where the north and south poles
        // currently are in local variables.  Compute the distance
        // between those x coordinates.  Move the poles that distance
        // in the x coordinate and 0 in the y distance.
    }

    // Return the north pole of the magnet
    public Pole getNorth () {
        return northPole;
    }

    // Return the south pole of the magnet
    public Pole getSouth()  {
        return southPole;
    }

    // Cause another magnet to move if it is close enough to cause
    // a magnet reaction.
    // Parameter:  other - the magnet to check for a reaction
    public void interact (Magnet other)  {
        // Check for attraction between the opposite poles of this
        // magnet and the other magnet.
        // Check for repulsion between similar poles of this
        // magnet and the other magnet.
    }
}
```