CS 051 Homework Laboratory # 4
Boxball

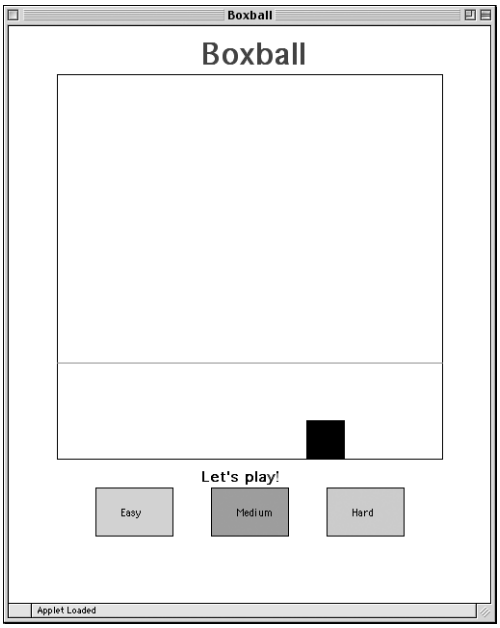
Objective: To gain experience defining constructor and method parameters and using active objects.

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:



Design of the program.

For this lab you will need to define ta Box class, along with a main program boxBall (an object that inherits graphicApplication). You are expected to come to lab with a design for the class and object 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.

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.

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.

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 500 by 600 when creating the application.)

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" button, 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 program 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 Boxball program 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, width:= (or you could name it setSize if you prefer), 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 width:= to accomplish this command.

After writing the width:= method, test it by modifying the onMousePress 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.

Note: Some of you may be tempted to skip the box class and just use a FramedRect, putting the other behavior elsewhere. Please do not do this, as writing the Box class will help you practice some of the kinds of code that will be needed in future labs.

Part 3: Dropping a ball We will be using the Animation module to move the ball smoothly down the screen. To get access, at the top of your program you will need:

```
import "Animation" as animator
```

Now you will be able to get access to the variants of while loops that include pauses for the animation.

The player will create a ball by pressing the mouse button in the playing area above the starting line. When the click is detected, a new ball will be created where the user pressed the mouse on the screen. The while()pausing()do() method of animator can then move it down the screen. Recall that

```
animator.while{cond}pausing(pauseTime)do{
      repeatingCode
}
```

executes repeatingCode repeatedly checking cond each time when it starts the loop and pauses by pauseTime after each iteration. When cond is false, the loop terminates.

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. On each iteration of the while loop, the ball should move a small distance, say 4 units. Then it should wait a short time. A pause time of 30 milliseconds is nearly undetectable by the human eye. 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. 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. Be sure to use the finally clause to remove the ball from the canvas.

Make sure that the ball is dropped only 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. As you test it, 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 left and right that give the positions of the edges of the box.

You will need to determine whether the ball is in the box only after the ball has reached the bottom of the court, as the ball could move while the ball is falling (can you figure out when that might happen?). However, the code that follows the method request of while()pausing()do() will generally be executed before the while loop has terminated. As a result, we will have to use a slightly different method from animator, the while()pausing()do() finally () method. Remember that when the system evaluates while{cond}pausing()do{repeatingCode}finally{endCode} the repeatingCode will be executed repeatedly as before, but when cond is false, Grace will execute the endCode. As a result, you will need to put your code to check whether the ball has fallen into the box in the finally block. Also in the finally clause you should add Grace code to remove the ball from the canvas.

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!". It should construct a Text object that displays a greeting message.

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

Finally, use the random number method randomIntFrom()to() 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. It doesn't need any parameters, as the box itself will decide *where* it moves.

When the box is narrow, it's left 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

As usual this assignment will be due at 11 pm on Monday evening.

Table 1: Grading Guidelines

Value	Feature
	Design preparation (3 pts total)
1 pt.	Boxball class
1 pt.	Box class
1 pt.	Ball class
	Readability (7 pts total)
2 pts.	Descriptive comments
2 pts.	Good names
2 pts.	Good use of constants
1 pt.	Appropriate formatting
	Code 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

Submitting Your Work

Before submitting your work, make sure that each of the .grace 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. In particular, be sure it conforms to the guidelines in the CS 51 Style Guide.

Turn in your project the same way as in past weeks by dragging your program folder to the dropoff folder.

MagnetGame Design

```
// Program that creates two magnets that may be dragged around the screen
// or clicked on to flip the polarity. When magnets are near each other
// they are attracted or repelled.
// @author Jane Cool

dialect "objectdrawDialect"
import "poleModule" as poleMod

type Pole = poleMod.Pole

type Magnet = poleMod.Magnet

def pole = poleMod.pole

def magnetGame: GraphicApplication = object {
    inherits graphicApplication.size(600,600)
    // where magnet was last
    var lastPoint: Point

    // create magnets
    var movingMagnet :Magnet := // magnet at 200 @ 100
    var restingMagnet :Magnet := // magnet at 200 @ 400

    // is a magnet being dragged
    var dragging:Boolean := false

    // Determine which, if any, magnet mouse is on when pressed
    // dragging set to true iff mouse pressed on a magnet
    // mousePoint is location where mouse is pressed.
    method onMousePress(mousePoint:Point)->Done{
        // remember point in lastPoint for dragging later
        // if point in movingMagnet then set dragging true
        // else if point in restingMagnet, set dragging true
        //       and swap names of two magnets so one containing
        //       point is now movingMagnet
        // else set dragging false
    }

    // Drag selected magnet as far as mouse is dragged
    // mousePoint -- location where mouse dragged to
    method onMouseDrag(mousePoint:Point)->Done{
        // if dragging a magnet then
        //     move movingMagnet by distance from lastPoint
        //     test its interaction with restingMagnet;
        //     remember point in lastPoint;
        }
    }

    // Flip magnet if it is clicked on
    // mousePoint -- location where mouse is clicked
```

```
    method onMouseClick(mousePoint:Point)->Done{
        // if clicked on the movingMagnet then flip it
        // & test its interaction with the restingMagnet);
    }
}
```

Magnet Design

```
// Class representing a magnet drawn on the screen with north and south poles.
// Methods are provided that allow it to be moved and determine how it interacts
// with other magnets. Also uses Pole class.
// @author Jane Cool

class magnet.at(locn': Point)on(canvas: DrawingCanvas) -> MagnetType {

    // Dimenstions of magnet
    def width:Number is public = 150
    def height:Number is public = 50

    // solid body of magnet and its outline
    def box: Graphic2D = // filled rect at locn' with size width x height

    def boxFrame: Graphic2D = // framed rect as above

    // the two poles
    def southPole:Pole is public = ...

    def northPole:Pole is public = ...

    // return the upper-left coordinates of the magnet
    method location -> Point {
        // return location of box;
    }

    // move the magnet by dx to right and dy down
    method moveBy(dx:Number,dy:Number)->Done {
        // move all the pieces by dx and dy
    }

    // move magnet to newLocn
    method moveTo(newLocn:Point)->Done {
        // let dx be distance in x direction from current location to newLocn
        // & dy be distance in y direction from current location to point

        // move this magnet by dx, dy
    }

    // Does this magnet contain locn
    method contains(locn:Point)->Boolean {
        // return whether the box contains the point
    }
```

```
    // Swaps the north and south poles of the magnet.
    method flip -> Done {
        // calculate difference d between north & south poles
        // move northPole by d in x direction and southPole by -d
    }

    // determine interactions of this magnet with the other magnet,
    // moving the other magnet if attracted or repelled
    // other - the other magnet being attracted or repelled
    method interact(other:Magnet)->Done {
        // check whether both pairs of opposite poles attract
        // and whether both pairs of same poles repel each other
    }

}
```

Boxball.java

```
dialect "objectdrawDialect"
import "Animation" as animator

// --- WHAT DOES IT DO?
// author -----

def boxBall: GraphicApplication = object {
    inherits graphicApplication . size (500,600)

    // ???????

    // ???????
    // parameter mousePoint -- ?????
    method onMousePress(mousePoint: Point)->Done{

    }
}
```

Box.java

```
class box. size ( size ': Number)top(top':Number)
            range( leftExtreme ': Number,rightExtreme':Number)on(canvas':DrawingCanvas) -> Box {

    // ????

    // moveBox to randomly chosen new location
    method moveBox -> Done {

    }

    // reset width of box
    method width:=(boxSize: Number)->Done {

    }

    // return x coordinate of left edge of box
    method left->Number {theBox.x} {


    }

    // return x coordinate of right edge of box
    method right->Number {left+theBox.width} {

    }

}
```