CS 051 Homework Laboratory # 4
Boxball

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

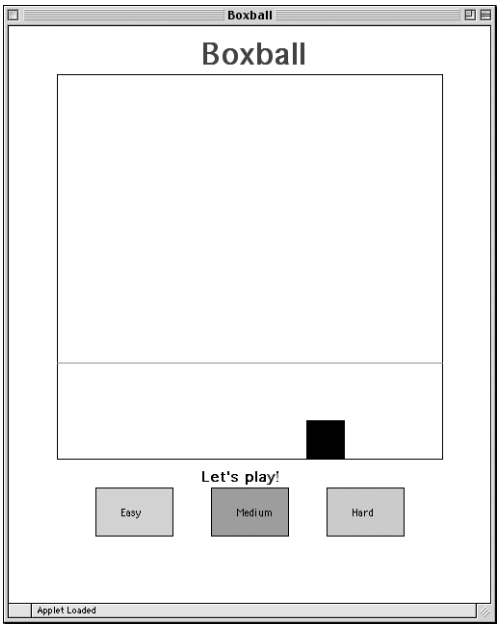
—

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:



To start, drag the BoxBall starter folder from the file server and then upload the starter code into FireFox. You can also look at the starter code by clicking on http://www.cs.pomona.edu/classes/cs051G/labs/boxball/BoxBall.grace.

Design of the program.

For this lab you will need to define a `box` class for the box that moves back and forth, a `ball` class for the falling ball, and a `boxball` object for the main program that `inherits graphicApplication`. You

are expected to come to lab with a design for each 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 the magnet program from last week's lab at the end of this handout.

The `boxball` program/object 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 and the box should change its width appropriately. 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 be removed from the canvas. If the ball lands in the box, the box should be instructed to choose and 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.grace`. (The canvas itself should be set to be 500 by 600 pixels when inheriting from `graphicApplication`.)

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` class in order to construct it properly. First, you need to tell the box where it might 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? All of those will need to be provided to the `box` class when it creates a new `Box`.

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` object 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 initialization code for the `box` class, you should go back to the `boxball` code 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:=(...)`, 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 `width:=` method, test it by modifying the `onMouseClick` method in the `boxball` program. 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 coding that will be needed in future labs.

Part 3: Dropping a ball 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 falling ball should be constructed using class `fallingBall` so that its center is at the point where the user clicked. It should also start it moving down the screen. To do so, it will call a `start` method, which will cause the corresponding method in `fallingBall` to execute.

You need to think carefully about what information a ball needs to know to construct itself properly and be able to check at the end whether it was inside the box. 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 using the method `while{}pausing()do{}finally{}` of `animator`. We have provided a skeletal 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 `while{}pausing()do{}finally{}` will accomplish this. On each iteration of the while loop, the ball should move a small distance, say 4 pixels. Then it should wait a short time. The pause method call that we provided in the starter tells the ball to wait 30 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. You may leave the `finally` clause empty for now. We will come back to it in part 4.

Once you have written the `fallingBall` class and the `while` loop, you should test them. Return to your `boxBall` program, 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` class 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` object 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 `fallingBall` class as a parameter so that the ball can change the message appropriately when it

hits or misses the box. (Note that while we have an `overlaps` method for graphics objects, we done not have a method that determines whether an object is entirely contained in another unless the first is just a `Point`. Hence you will need to figure out how to do this.

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`. It will need no parameters as it is responsible for randomly choosing the new location and moving the box there.

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.

By the way, some of you may be tempted to figure out whether the ball will land in the box by checking at the moment the ball is dropped – anticipating that the ball will fall straight down. Unfortunately, that won't always work if multiple balls are falling. The problem is that the first might get in the box – resulting in the box moving – all while the second is falling. The bottom line: Don't check whether the ball is in the box until the ball falls down all of the way.

Due Dates

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

Submitting Your Work

Before submitting your work, make sure that your .grace file 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. In particular make sure your indentation is consistent.

Turn in your project the same as in past weeks. Change the name of the folder to `LabXX_LastNameFirstName`" where XX is the number of the lab (e.g. the first lab will be 01 note the leading zero) and LastNameFirstName is replaced with your own last and first name. Once you have renamed the folder, please drag it to the dropoff folder.

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.

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

MagnetGame Design

```
// Class 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 "objectdraw"

// type info omitted for Pole and Magnet

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

   // Dimensions of magnet.  Declaring them public generates methods to access them
   def width:Number is public = 150
   def height:Number is public = 50

   // solid body of magnet and its outline
   def box: Graphic2D = filledRect.at(locn')size(width,height)on(canvas)

   def boxFrame: Graphic2D = framedRect.at(locn')size(width,height)on(canvas)

   // return location of upper-left corner of magnet
   method location -> Point {
      // pass along location of box
   }

   // locations of north and south poles relative to locn'
   def northLocn:Point =  (locn'.x+width-height/2) @ (locn'.y+height/2 + 5)
   def southLocn:Point =  (locn'.x+height/2) @ (locn'.y+height/2 + 5)

   // north and south poles.  Declaring them public gives accessor method
   def southPole:Pole is public =
                pole.inside(self)at(southLocn)isNorth(false)on(canvas)

   def northPole:Pole is public =
                pole.inside(self)at(northLocn)isNorth(true)on(canvas)

   // set box to be red

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

   // move magnet to newLocn
   method moveTo(newLocn:Point)->Done {
```

```
      // Figure out how far to move (e.g. newLocn.x-box.x & newlocn.y - box.y)
      // then moveBy that amount
   }

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

   // reverse the location of north and south poles
   method flip -> Done {
      // interchange north and south poles by calculating distance between them
      // and then moving them appropriately
   }

   // determine whether this magnet close enough to other to attract or repel
   method interact(other:Magnet)->Done {
      // check to see if opposite poles attract (use tryAttract)
      // or similar poles repel (using tryRepel)
   }
}


// Program that draws & interacts with magnets.
def magnetGame: GraphicApplication = object {
   inherits graphicApplication.size(600,600)

   // Where mouse was before last drag
   var lastPoint: Point

   // two magnets.  First is always the one being dragged
   var movingMagnet :Magnet := magnet.at(200 @ 100)on(canvas)
   var restingMagnet :Magnet := magnet.at(200 @ 400)on(canvas)

   // whether or not a magnet is being dragged
   var dragging:Boolean := false

   // Determine which magnet, if any, mouse is pressed on.
   // Call it movingMagnet and the other restingMagnet
   method onMousePress(mousePoint:Point)->Done{
      // remember where mouse pressed in lastPoint
      // if movingMagnet contains the mousePoint then remember that dragging
      //  Otherwise if other magnet contains mousePoint then remember dragging
      // and swap the names of the magnets
      // otherwise remember that not dragging a magnet
   }

   // if dragging, then drag movingMagnet as far as mouse moves
```

```
   method onMouseDrag(mousePoint:Point)->Done{
      // if dragging a magnet then move it & see if the magnets now interact
   }

   // flip the magnet that is clicked on
   method onMouseClick(mousePoint:Point)->Done{
      // not required in the design, but ...
      // flip the magnet being dragged
      // check to see if the magnets interact
   }

   // always required to display window and start graphics
   startGraphics
}
```