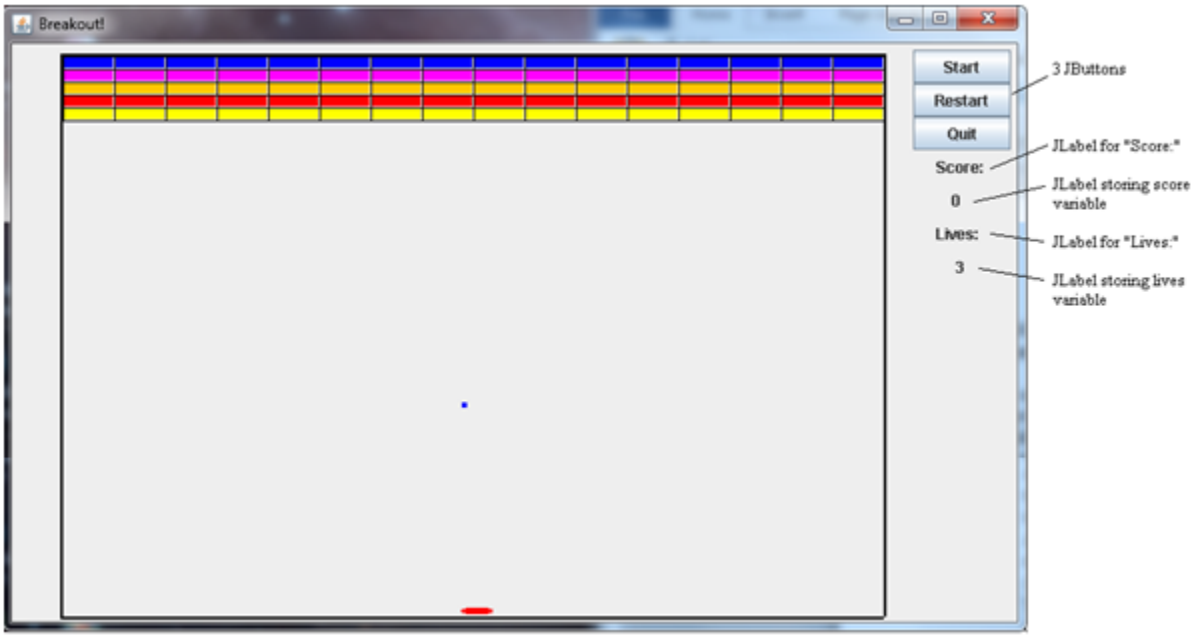
CSC 360 Programming assignment #7
Due Date: Friday, October 31

In this assignment, you will implement the game of Breakout utilizing several classes. The game uses a keyboard to control a paddle that moves near the bottom of the screen. The object is to hit the ball with the paddle such that the ball then hits one or more of the bricks along the top. The ball bounces off of bricks, the paddle and walls. If you miss the ball with the paddle, the ball is "lost". You have a limited number of lives (for instance 3). Implementing this game will require that you implement the following:

- A ball class which will contain its own x, y coordinate and motion and methods to move the ball, determine if the ball hits something, and draw the ball onto a Graphics object
- A paddle class which will contain its own x coordinate and motion (in the x-direction) and methods to move the paddle, determine if the paddle has hit either border or the ball, and draw the paddle onto a Graphics object
- A brick class which will contain the brick's x, y coordinate, Color, value and visibility, and methods to determine if the brick has been hit by the ball and to draw it on a Graphics object if it is visible
- A main class which will create a JFrame, keep track of the score and the number of lives remaining, a Timer and create two inner class instances of JPanels to place onto the JFrame
- One JPanel will implement ActionListener for JButtons and contain 3 JButtons and 4 JLabels. The ActionListener will control whether the Timer should be running or stopped, and display the current score and lives in the JLabels as they change.
- The other JPanel will implement ActionListener for a Timer and KeyListener to control the paddle's movement, and a paintComponent method to draw the game area (Bricks, Paddle, Ball).



You do not need the Random class for this program unless you want the ball to bounce randomly or start in a random direction and/or location. The UML descriptions of the classes follow. NOTE: while you should stick to these specifications, you should feel free to add enhancements to the game as desired such as having multiple levels with different patterns of bricks, changing the Paddle's size as the game goes on (making it shorter) or making the Ball travel faster, etc. Have fun with the assignment and if your game is not perfect, it will not cause you to get a lower grade as long as you implement the classes as specified.

Ball
-bx, by, bdx, bdy: int // bdx, bdy is the motion of the ball (the amount of change per movement)
-leftBorder, upperBorder, rightBorder, lowerBorder: int // keep track of the boundary of the area*
+Ball(leftBorder, upperBorder, rightBorder, lowerBorder: int) (or a no-arg constructor) *
+getX( ): int // accessors
+getY( ): int
+move( ): void // move the Ball to a new bx, by position based on bdx, bdy
+change(newBDX: int, newBDY: int): void // change bdx, bdy after a collision
+change(change int): void // change bdx, bdy after a collision
+draw(g: Graphics): void // draw the Ball on g at bx, by

* Either pass the boundaries of the playing area to the constructor to initialize the border variables or hard-code them (your choice). If you hard-code them, then implement a no-arg constructor otherwise implement a 4-arg constructor. The constructor should initialize bx, by to some starting point (say 2/3s between the top and bottom border and in the center), bdx, bdy to some initial value (such as 1 each which would move the Ball down to the right), or to random values (make sure bdy is never 0 as this would move the Ball horizontally but not vertically), and the borders. The borders are used to determine if the Ball should rebound from a side or top.

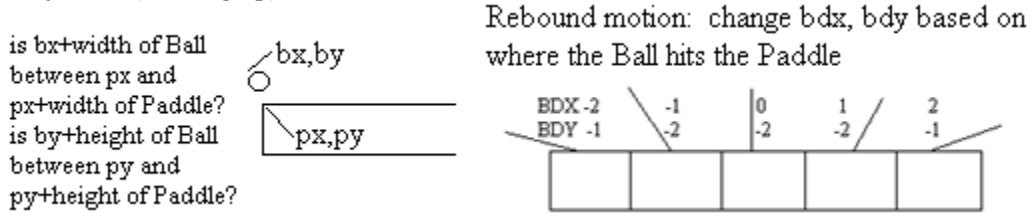
The move method will move the Ball: `bx+=bdx; by+=bdy;`. Thus, the Ball moves by the values of bdx and bdy. For instance, if bx=200, by=300, bdx=-1 and bdy=2 then bx becomes 199 and by becomes 202 (the Ball moved a little to the left and down). The move method should then check to see if the ball has hit a left, right or upper border. If so, "deflect the ball" by changing bdx or bdy or both. If the Ball hits the left/right border, change bdx only and if it hits the top border, change bdy only. You can change the value by multiplying it by -1 (e.g., `bdx*=-1;`) which is a simple rebound, or by changing it to a random value between -2 and +2 (a larger value for bdx and bdy will make the ball travel more (faster) and so may make the game harder). Make sure bdy is not 0.

We test to see if the Ball hits the Paddle or a Brick elsewhere. We also test to see if the Ball passes the Paddle elsewhere. Those locations will need to rebound the Ball also, so we provide change methods. These two methods cause the ball to change direction by altering its bdx and bdy values. The one-arg change method changes bdx to the parameter and multiplies bdy by -1. For draw, draw a fillOval(bx, by, 5, 5) in the Color of your choice (I chose blue). You can make the Ball larger if desired (do not make it smaller though, it might be too hard to see).

Paddle
-px, py, pdx: int // the paddle's location and motion, note no pdy (it never moves up/down)
-leftBorder, rightBorder: int // keep track of the boundary of the area*
+Paddle(leftBorder, rightBorder: int) (or a no-arg constructor)*
+collide(b: Ball): void // does Ball b collide with this Paddle? If so, tell the ball how to rebound
+move( ): void // move the Paddle left or right
+change(x: int): void // change pdx because of a key press action
+draw(g: Graphics): void // draw the Paddle onto the Graphics object

* - as with Ball, Paddle can either be told the boundaries (just left/right) through parameters to the constructor or these can be hard-coded. Use either a 2-arg or no-arg constructor. The collide method must determine if the Ball has hit the Paddle and where. This requires testing the bottom edge of the Ball against the top edge of the Paddle and both the right edge of the Ball against the left edge of the Paddle

and the left edge of the Ball against the right edge of the Paddle. If we determine that the Ball falls within these extremes, then there is a collision. Since collide receives the Ball b, you will have to get b's bx, by coordinates to test them to px, py. Upon a collision, tell b how to change itself by using the 2-arg change method. The change should be based on where the Ball hits the Paddle. Break the Paddle into 5 regions as shown below. The values shown are the new bdx and bdy values for the Ball. For instance, hitting the Paddle along the rightmost side would cause the Ball to have a bdx value of 2 (heading quickly to the right) and bdy of -1 (heading up).



Brick	
-x, y: int	// the upper left hand corner of this brick
-value: int	// the amount of score the player earns for hitting this brick
-visibility: boolean	// if not yet hit, it is visible and thus still in play, once hit this becomes false
-color: Color	// the Color to draw this brick
+Brick(x, y, v: int)	// pass in the x,y coordinate and value, value is used to determine Color
+isVisible( ): boolean	// accessor
+collides(b: Ball): int	// see below
+draw(g: Graphics): void	// if this Brick is visible, draw it at the x,y location using its color onto the // Graphic object, bricks should be 40x10 (in my game, I use drawRect 40x10 to // draw a black empty brick and then a 38x8 fillRect of this Brick's color inside // the drawRect to give every Brick a black border

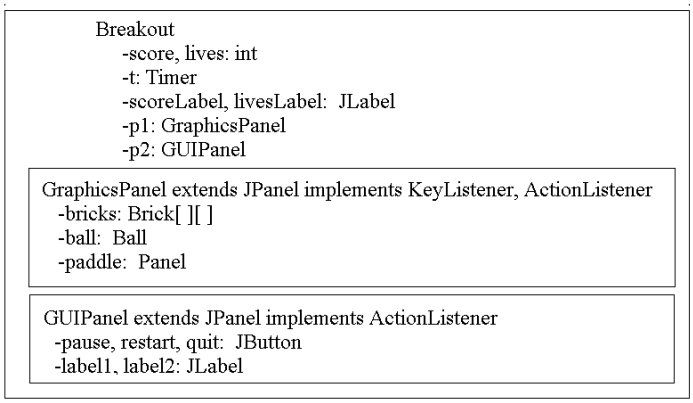
The collides method does most of the work in this class. Given the Ball b, does it collide with this Brick? This will work as follows:

If this Brick is visible then
Get b's bx,by values
Does b hit the Brick (is bx,by touching an edge of this Brick?), if so determine which part of the Brick is being hit (divide the Brick into 5 sections like the Paddle, or 3 sections if desired), store the new BDX value in temp (this will be between -2 and 2 if you divide the Brick into 5 sections, or -1 to 1 if you divide it into 3 sections). Call `b.change(temp);`, set visible to false (this Brick is no longer visible, and return value.
If b doesn't hit this Brick, return 0.

The UML (not including methods) for the final class, Breakout, is shown at the top of the next page. This class will have just a main method but will also have its own instance data and two nested inner classes for two JPanels. One JPanel will be for Graphics and the other will have the GUI features.

GUIPanel: its only method is actionPerformed. This method will have to determine which JButton led to the ActionEvent (use `e.getSource( )`). For quit, do `System.exit(0);` to immediately end the program. For restart, invoke `p1.restart( );` (described below). The pause button will be used to start, pause and resume the game. The JButton will initially say "Start" and once started, will change to "Pause". If clicked again, it pauses the game and should now read "Resume". Clicking it will resume the game and change the JButton to "Pause". To change the text of the JButton, use

`pause.setText("...");`. To pause the game, use `t.stop();` (stop the Timer) and to start or resume the game, use `t.start( );` Note that if you use a JButton, the program's focus shifts from the GraphicsPanel to the GUIPanel. We need to regain the focus in the GraphicsPanel for the KeyListener to work. Add to your actionPerformed method (of GUIPanel) the following: `p1.requestFocusInWindow( );`



GraphicsPanel: The constructor should just call the method restart. This method will instantiate and initialize all of the game's variables. This method is also called from the GUIPanel's actionPerformed when the restart button is clicked. Note that this method will not start the Timer as this will be handled in GUIPanel's actionPerformed. Otherwise, this method will instantiate the Ball and Paddle, initialize score to 0, lives to 3 (or some other value) and instantiate the scoreLabel, livesLabel, and t (Timer, which I set to a 10 millisecond delay). Also add the KeyListener to this class. Finally, instantiate the Bricks array and then initialize each individual Brick. I used the following code for this.

```
bricks=new Brick[5][16];
for(int i=0;i<5;i++)                    // 5 rows
   for(int j=0;j<16;j++)                // 16 bricks per row
     bricks[i][j]=new Brick((j+1)*40, (i+1)*10, (6-(i+1))*10);
```

The x,y coordinate of each Brick is based on the values of i, j. The Brick's value is (6-(row+1))*10.

You will have to implement one of the three KeyListener methods. I chose keyPressed. If the left arrow key is pressed, change the paddle's pdx value to be pdx-1, if the right arrow key is pressed, change the paddle's pdx value to be pdx+1 and if the down arrow key is pressed, change the paddle's pdx value to 0 (this stops the paddle). Note in the Paddle class, change should make sure the Paddle is never traveling too fast. I used -3 to +3 as the limits. If in changing pdx it becomes < -3 I reset it to -3 and if in changing pdx it becomes > +3 I reset it to +3. This limits its speed to only slightly greater than the ball can travel (at most, the ball is traveling -2 to +2 in the x-direction).

The actionPerformed method is called whenever the Timer generates an ActionEvent (every 10 milliseconds if you use 10 as the delay). Here, call `ball.move( );` and `paddle.move( );`. If the ball, after moving, has a y value > the lower limit of the playing area, the user loses a life, do `lives--;` `livesLabel.setText("  "+lives);` and `ball=new Ball(…);` to start the ball over. If lives becomes 0, stop the Timer. If the Ball has not gone below the Paddle, test to see if the Ball collides with the Paddle or any of the Bricks (use a nested for loop). Remember that the Ball can only collide with a visible Brick. If you did these classes correctly, the Ball will rebound based on code implemented in the Ball, Paddle and Brick classes. If the Ball hits a Brick, you will get back the amount of the Brick, add this to score and set the text of the scoreLabel to indicate the new score. The last thing actionPerformed should do is call `repaint( );` to redraw the game, reflecting the latest movements.

In paintComponent, execute `super.paintComponent(g);` and then draw a black border around the playing area. Next, do `ball.draw(g);` `paddle.draw(g);` and using a nested for loop, draw all the bricks (`bricks[i][j].draw(g);`). Finally, if lives is 0, use drawStirng to output "GAME OVER" somewhere in the center of the playing area.

Hand in the four classes (including the two inner classes) and a screenshot of your game.