Sarai Gould & Laura Cruz-Albrecht
CS 106A

Assignment 3
July 10, 2019

Assignment #3—Breakout
Due: 10AM PST on Thursday, July 18th

This assignment may be done in pairs (which is optional, not required)

Based on handouts by Marty Stepp, Mehran Sahami, Eric Roberts, and Chris Piech with modifications by Brahm Capoor

The purpose of this assignment is to practice creating graphical programs, using concepts such as events, animation, and instance variables. You will first implement **two Sandcastles** as warm-up problems. Then, your main task will be to implement the classic arcade game of **Breakout**, an animated game described in more detail below.

Note that this assignment may be done in **pairs**, or may be done individually. **You may only pair up with someone in the same section time and location**. If you would like to work with a partner but don't have one, you can try to meet one in your section. If you work as a pair, **comment both members' names** on top of every .java file. **Only one** of you should submit the assignment; do not turn in two copies.

In general, limit yourself to using Java syntax taught in lecture, and the parts of the textbook we have read, up through the week of the release of this assignment (July 11). You may, however, use material covered in class past this date for any optional extensions. If you would like to implement any extensions, please *implement them in a separate file*, such as `BreakoutExtra.java`. Clearly comment at the top of this file what extensions you have implemented. Instructions on how to add files to the starter project are listed in the FAQ of the Eclipse page on the course website.

Sandcastles

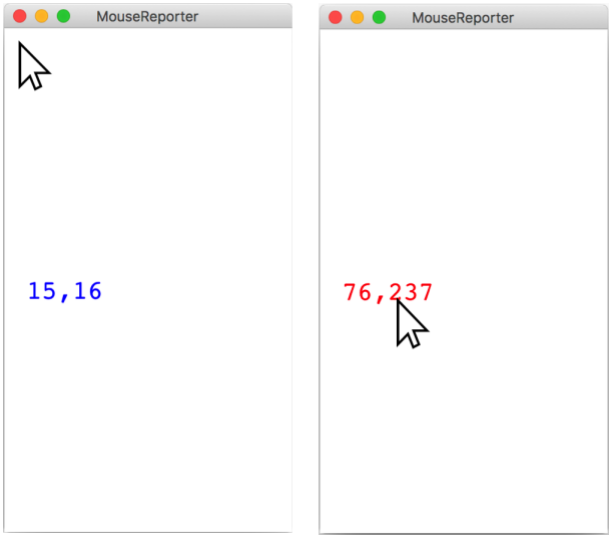
Sandcastle #1: Prime Checker

A key challenge when writing large programs like Breakout is how best to decompose our solutions into manageable and effectively-implemented methods. In order to practice this skill, you'll begin this assignment by writing a short method that takes in a positive integer greater than 1 as an input and returns a boolean indicating whether or not that integer is prime.

As a reminder, a prime number that is defined such that its only factors are 1 and itself. In `PrimeChecker.java`, we provide a `run` method that tests whether or not a series of numbers are prime. Your job is to implement the `isPrime` method to check whether or not the number is prime. **Make sure to submit your PrimeChecker with Breakout.**

Sandcastle #2: Mouse Reporter

To get you warmed up, you will next write a minimal program that leverages the essential concepts needed for Breakout. Write a `MouseReporter` that creates a GLabel on the left side of the screen. When the mouse is moved the label is updated to display the current x, y location of the mouse. If the mouse is touching the label it should turn **red**, otherwise it should be **blue**.



You should take advantage of the **`setLabel`** method that can be called on a **`GLabel`**. If you look in Chapter 9 (page 299) at the methods that are defined at the **`GraphicsProgram`** level, you will discover that there is a method

```
public GObject getElementAt(double x, double y)
```

that takes a position in the window and returns the graphical object at that location, if any. If there are no graphical objects that cover that position, **`getElementAt`** returns the special constant **`null`**. If there is more than one, **`getElementAt`** always chooses the one closest to the top of the stack, which is the one that appears to be in front on the display. The starter code for **`MouseReporter`** stores the label as an instance variable and adds it to the screen. **Make sure to submit your MouseReporter with Breakout.**

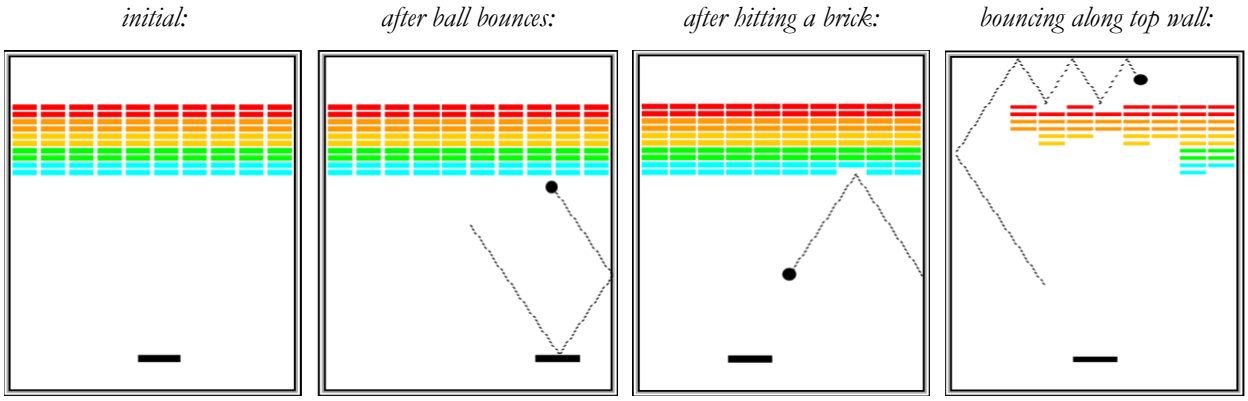
Breakout

Your main task is to write the classic arcade game of Breakout, which was invented by Steve Wozniak before he founded Apple with Steve Jobs. It is a large assignment, but entirely manageable as long as you follow the following pieces of advice:

- *Start as soon as possible.* This assignment is due in just over a week, which will be here before you know it. Don't wait until the last minute!
- *Implement the program in stages, as described in this handout.* Don't try to get everything working all at once. Implement the various pieces of the project one at a time and make sure that each one is working before you move on to the next phase.
- *Don't try to extend the program until you get the basic functionality working.* At the end of the handout, we suggest several ways in which you could optionally extend the game. Several of these are lots of fun. Don't start them, however, until the basic assignment is working. If you add extensions too early, you'll find that the debugging process gets really difficult.

Game Mechanics

In Breakout, the player controls a rectangular paddle that is in a fixed position in the vertical dimension but moves back and forth across the screen horizontally along with the mouse within the bounds of the screen. A ball moves about the rectangular world and bounces off of surfaces it hits. The world is also filled with rows of rectangular bricks that can be cleared from the screen if the ball collides with them. The goal is to clear all bricks.



The player has three lives, or *turns*. On each turn, a ball is launched from the center of the window toward the bottom of the screen at a random angle. That ball bounces off the paddle and the walls of the world. The second of the figures above shows the ball's path after two bounces, one off the paddle and one off the right wall. *(Note that the dotted line is there just to illustrate the ball's path and won't appear on the screen.)*

When the ball collides with a brick, the ball bounces as normal, but the brick disappears, as illustrated in the third of the figures above. This diagram also shows the player moving the paddle leftward to line it up with the oncoming ball.

A **turn ends** when one of two conditions occurs:

1) The ball hits the lower wall, which means that the player must have missed it with the paddle. In this case, the turn ends and the next ball is served if the player has any turns left. If not, the player loses.
2) The last brick is eliminated. In this case, the player wins, and the game ends immediately.

After all the bricks in a particular column have been cleared, a path will open to the top wall. When this situation occurs, the ball will often bounce back and forth several times between the top wall and the upper line of bricks without the user ever having to worry about hitting the ball with the paddle. This condition is called "breaking out", as shown in the farthest-right of the figures above, and gives meaning to the name of the game. That ball will go on to clear several more bricks before it comes back down an open channel.

It is important to note that, even though breaking out is a very exciting part of the player's experience, you don't have to do anything special in your program to make it happen. The game is simply operating by the same rules it always applies: bouncing off walls, clearing bricks, and otherwise obeying the laws of physics.

The Starter File

The starter project for this assignment has a little more in it than it has in the past, but none of the important parts of the program. The starting contents of the `Breakout.java` file appear in Figure 1 (on the next page). This file takes care of the following details:

- It includes the imports you will need for writing the game.
- It defines the named constants that control the game parameters, such as the dimensions of the various objects. Your code should use these constants internally so that changing them in your file changes the behavior of your program accordingly.

Success in this assignment will depend on breaking up the problem into manageable pieces and getting each one working before you move on to the next. The next few sections describe a reasonable staged approach to the problem.

Figure 1. **The `Breakout.java` starter file has many constants**

```
public class Breakout extends GraphicsProgram {

      // Dimensions of the canvas, in pixels
      // These should be used when setting up the initial size of the game,
      // but in later calculations you should use getWidth() and getHeight()
      // rather than these constants for accurate size information.
      public static final double CANVAS_WIDTH = 420;
      public static final double CANVAS_HEIGHT = 600;

      // Number of bricks in each row
      public static final int NBRICK_COLUMNS = 10;

      // Number of rows of bricks
      public static final int NBRICK_ROWS = 10;

      // Separation between neighboring bricks, in pixels
      public static final double BRICK_SEP = 4;

      // Width of each brick, in pixels
      public static final double BRICK_WIDTH = Math.floor(
                  (CANVAS_WIDTH - (NBRICK_COLUMNS + 1.0) * BRICK_SEP) /
                  NBRICK_COLUMNS);

      // Height of each brick, in pixels
      public static final double BRICK_HEIGHT = 8;

      // Offset of the top brick row from the top, in pixels
      public static final double BRICK_Y_OFFSET = 70;

      // Dimensions of the paddle
      public static final double PADDLE_WIDTH = 60;
      public static final double PADDLE_HEIGHT = 10;

      // Offset of the paddle up from the bottom
      public static final double PADDLE_Y_OFFSET = 30;

      // Radius of the ball in pixels
      public static final double BALL_RADIUS = 10;

      // The ball's vertical velocity.
      public static final double VELOCITY_Y = 3.0;

      // The ball's minimum and maximum horizontal velocity; the bounds of the
      // initial random velocity that you should choose (randomly +/-).
      public static final double VELOCITY_X_MIN = 1.0;
      public static final double VELOCITY_X_MAX = 3.0;

      // Animation delay or pause time between ball moves (ms)
      public static final double DELAY = 1000.0 / 60.0;

      // Number of turns
      public static final int NTURNS = 3;

      ...

}
```

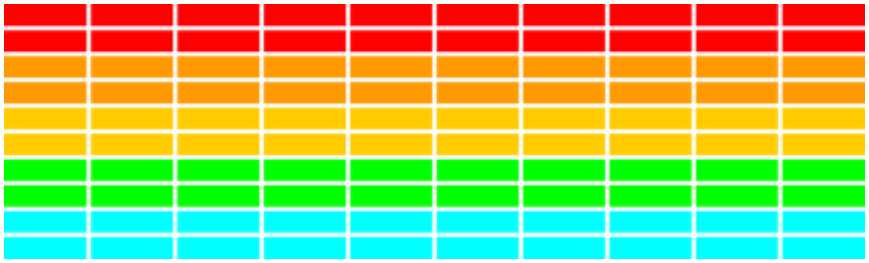
Approach

Since this is a tough program, we strongly recommend that you develop and test it in several stages, always making sure that you have a program that compiles and runs properly after each stage. Here are the stages we suggest, each discussed in more detail in the rest of the handout:

1) Bricks
2) Paddle
3) Ball and Bouncing
4) Collisions
5) Turns, End of Game, and Final Touches

Stage 1: Bricks

Before you start playing the game, you have to set up the various pieces. Thus, it probably makes sense to implement the **`run`** method as two method calls: one that sets up the game and one that plays it. An important part of the setup, and our first suggested task, consists of creating the rows of bricks at the top of the game, which look like this:



The number, dimensions, and spacing of the bricks are specified using named constants in the starter file, as is the distance from the top of the window to the first line of bricks. The only value you need to compute is the *x* coordinate of the first column, which should be chosen so that the bricks are centered in the window, with the leftover space divided equally on the left and right sides.

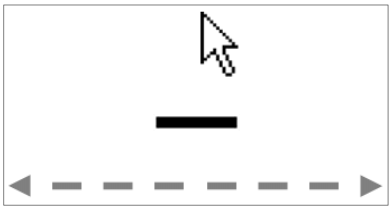
Each pair of rows has a given color; the colors run in the following sequence: **`Color.RED`**, **`ORANGE`**, **`YELLOW`**, **`GREEN`**, **`CYAN`**. Do not assume that there will be an even number of rows, nor that there will be no more than 10 rows.

You've already drawn 2D grids of objects (remember the Checkerboard problem from lecture?). The part that is a touch more difficult is that you need to appropriately position and color the bricks.

Relevant constants: `NBRICK_COLUMNS`, `NBRICK_ROWS`, `BRICK_SEP`, `BRICK_WIDTH`, `BRICK_HEIGHT`, `BRICK_Y_OFFSET`

Stage 2: Paddle

Our next suggested task is to create the paddle. In a sense, this is easier than the bricks, since there is only one paddle, which is a filled **`GRect`**. You must set its size to be **`PADDLE_WIDTH`** by **`PADDLE_HEIGHT`** and y-position relative to the bottom of the window to be **`PADDLE_Y_OFFSET`**. Note that **`PADDLE_Y_OFFSET`** is the distance between the bottom of the screen and the *bottom* of the paddle.

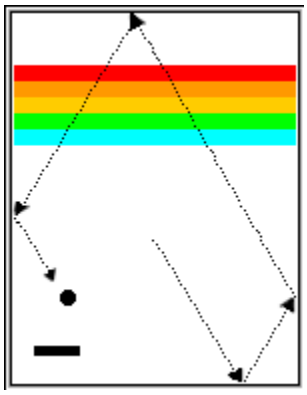


The paddle horizontally follows the mouse as it moves onscreen; specifically, you must make the horizontal center of the paddle follow the mouse. Mouse tracking uses events discussed in Chapter 10 of the textbook, and in lecture on 7/10. You only have to pay attention to the *x* coordinate of the mouse because the paddle's *y* position is fixed. Also, do not allow any part of the paddle to move off the edge of the screen. Check to see whether the x-coordinate of the mouse extends beyond the screen boundary and ensure that the entire paddle is visible in the window.

Relevant constants: `PADDLE_WIDTH, PADDLE_HEIGHT, PADDLE_Y_OFFSET`

Stage 3: Ball + Bouncing

Now let's make the ball and get it to bounce around the screen (for now ignoring brick and paddle collisions, or going off of the bottom).



At one level, creating the ball is easy, given that it's just a filled **`GOval`**. The interesting part lies in getting it to move and bounce appropriately. You are now past the "setup" phase and into the "play" phase of the game. To start, create a ball and put it in the center of the window. As you do so, keep in mind that the coordinates of the **`GOval`** do not specify the location of the center of the ball but rather its upper left corner. The mathematics is not any more difficult, but may be a bit less intuitive.

The program needs to keep track of the velocity of the ball, which consists of two separate components, which you will presumably declare as instance variables like this:

```
private double vx, vy;
```

You may make these private instance variables, as they will be used throughout your program and are useful "game state". The velocity components represent the change in position that occurs on each time step. Initially, the ball should be heading downward, and you might try a starting velocity of +3.0 for **`vy`** (remember that *y* values in Java increase as you move down the screen). The game would be boring if every ball took the same course, so you should choose the **`vx`** component randomly.

In line with our discussion of generating random numbers this week, you should make a random-number generator:

1. Declare an instance variable **rgen**, which will serve as a random-number generator:

   ```
   private RandomGenerator rgen = RandomGenerator.getInstance();
   ```

   Remember that instance variables are declared outside of any method but inside the class.

2. Initialize the **vx** variable as follows:

   ```
   vx = rgen.nextDouble(1.0, 3.0);
   if (rgen.nextBoolean(0.5)) vx = -vx;
   ```

   This code sets **vx** to be a random **double** in the range 1.0 to 3.0 and then makes it negative half the time. This strategy works much better for Breakout than calling

   ```
   nextDouble(-3.0, +3.0)
   ```

   which might generate a ball going more or less straight down. That would make life far too easy for the player.

Once you've done that, your next challenge is to get the ball to bounce around the world, ignoring entirely the paddle and the bricks. To do so, you need to check to see if the coordinates of the ball have gone beyond the boundary, taking into account that the ball has a nonzero size. Thus, to see if the ball has bounced off the right wall, you need to see whether the coordinate of the right edge of the ball has become greater than the width of the window; the other three directions are treated similarly. (Recall you can use **getWidth()** and **getHeight()** to find the game world's size.) For now, have the ball bounce off the bottom wall so that you can watch it make its path around the world. You can change that test later so that hitting the bottom wall signifies the end of a turn.

Computing what happens after a bounce is extremely simple. If a ball bounces off the top or bottom wall, all you need to do is reverse the sign of **vy**. Symmetrically, bounces off the side walls simply reverse the sign of **vx**.

Relevant constants: `BALL_RADIUS, VELOCITY_X_MIN, VELOCITY_X_MAX, VELOCITY_Y, DELAY`

Stage 4: Collisions

Now comes the interesting part. In order to make Breakout into a real game, you have to be able to tell whether the ball is colliding with another object in the window. As scientists often do, it helps to begin by making a simplifying assumption and then relaxing that assumption later. Suppose the ball were a single point rather than a circle. In that case, how could you tell whether it had collided with another object? What happens if you call

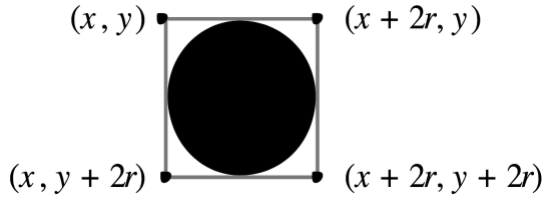
```
getElementAt(x, y)
```

where **x** and **y** are the coordinates of the ball? If the point **(x, y)** is underneath an object, this call returns the graphical object with which the ball has collided. If there are no objects at the point **(x, y)**, you'll get the value **null**.

So far, so good. But, unfortunately, the ball is not a single point. It occupies physical area and therefore may collide with something on the screen even though its center does not. The easiest thing to do—which is in fact typical of the simplifying assumptions made in real computer games—is to

check a few carefully chosen points on the outside of the ball and see whether any of those points has collided with anything. As soon as you find something at one of those points, you can declare that the ball has collided with that object.

In your implementation, the easiest thing to do is to check the four corner points on the square in which the ball is inscribed. Remember that a **`GOval`** is defined in terms of its bounding rectangle, so that if the upper left corner of the ball is at the point (x, y), the other corners will be at the locations shown in this diagram:



These points have the advantage of being outside the ball—which means that **`getElementAt`** can't return the ball itself—but nonetheless close enough to make it appear that collisions have occurred. Thus, for each of these four points, you need to:

1. Call **`getElementAt`** on that location to see whether anything is there.
2. If the value you get back is not **`null`**, then you need look no farther and can take that value as the **`GObject`** with which the collision occurred.
3. If **`getElementAt`** returns **`null`** for a particular corner, go on and try the next corner.
4. If you get through all four corners without finding a collision, then no collision exists.

It would be very useful to write this section of code as a separate method

```
private GObject getCollidingObject()
```

that returns the object involved in the collision, if any, and **`null`** otherwise. You could then use it in a declaration like

```
GObject collider = getCollidingObject();
```

which assigns that value to a variable called **`collider`**.

From here, the only remaining thing you need to do is decide what to do when a collision occurs. There are only two possibilities. First, the object you get back might be the paddle, which you can test by checking

```
if (collider == paddle) . . .
```

If it is the paddle, you need to bounce the ball so that it starts traveling up. If it isn't the paddle, the only other thing it might be is a brick, since those are the only other objects in the world. Once again, you need to cause a bounce in the vertical direction, but you also need to take the brick away. To do so, all you need to do is remove it from the screen by calling the **`remove`** method.

Relevant constants: `None`

Stage 5: Turns, End of Game, and Final Touches

We're almost there! There are, however, a few more important details you need to take into account:

- You've got to give the player 3 chances (constant: **`NTURNS`**) to break all the bricks. Every time the ball hits the bottom edge of the window, the player loses 1 turn. When the turn ends, if the player has more turns remaining, your program should re-launch the ball from the center of the window toward the bottom of the screen. The easiest way to do this is to call **`setLocation(x, y)`** `on` the ball to move it to the center of the window. Don't forget that the ball should receive a new random x velocity at the start of each turn.

- We recommend that before each round, you wait for the user to click the mouse by calling the **`waitForClick()`** method, which pauses until the user clicks the mouse.

- As part of tracking turns, you've got to take care of the case when the ball hits the bottom wall. In the prototype you've been building, the ball just bounces off this wall like all the others, but that makes the game pretty hard to lose. You've got to modify your loop structure so that it tests for hitting the bottom wall as one of its terminating conditions.

- You've got to check for the other terminating condition, which is hitting the last brick. How do you know when you've done so? Although there are other ways to do it, one of the easiest is to have your program keep track of the number of bricks remaining. Every time you hit one, subtract one from that counter. When the count reaches zero, you must be done. In terms of the requirements of the assignment, you can simply stop at that point, but it would be nice to give the player a little feedback that at least indicates whether the game was won or lost.

- You've got to experiment with the settings that control the speed of your program. How long should you pause in the loop that updates the ball? Do you need to change the velocity values to get better play action?

- You've got to test your program to see that it works. Play for a while and make sure that as many parts of it as you can check are working. A particular case to test: just before the ball is going to pass the paddle, move the paddle quickly so that it slides through the ball from the side. Does everything still work, or does your ball seem to get "glued" to the paddle? Why might this error occur? (think about how the ball collides with objects, and how this might explain the observed behavior) How can you fix it? (It is easier to test for this if you temporarily make the paddle taller by changing **`PADDLE_HEIGHT`**.)

Relevant constants: `NTURNS`

Optional Extra Features

There are many possibilities for optional extra features that you can add if you like, potentially for a small amount of extra credit. If you are going to do this, please *submit two versions of your program*: `Breakout.java` that meets all the assignment requirements, and a `BreakoutExtra.java` containing your extended version (see the FAQ on the Eclipse page for how to create a new file in your project). At the top of your extended file, in your comment header, you must **comment** what extra features you completed. Here are a few ideas of for possible extensions (of course, we encourage you to use your imagination to come up with other ideas as well):

- *Add sounds.* You might want to play a short bounce sound every time the ball collides with a brick or the paddle. This extension turns out to be very easy. The starter project contains an audio clip file called `bounce.au` that contains that sound. You can load the sound by writing

  ```
  AudioClip bounceClip = MediaTools.loadAudioClip("bounce.au");
  ```

  and later play it by calling

  ```
  bounceClip.play();
  ```

  The Java libraries do make some things easy.

- *Add messages.* The game is more playable if at the start it waits for the user to click the mouse before serving each ball and announces whether the player has won or lost at the end of the game. These are just `GLabel` objects that you can add and remove at the appropriate time.

- *Improve the user control over bounces.* The program gets rather boring if the only thing the player has to do is hit the ball. It is far more interesting, if the player can control the ball by hitting it at different parts of the paddle. The way the old arcade game worked was that the ball would bounce in both the *x* and *y* directions if you hit it on the edge of the paddle from which the ball was coming.

- *Add in the "kicker."* The arcade version of Breakout lured you in by starting off slowly. But, as soon as you thought you were getting the hang of things, the program sped up, making life just a bit more exciting. As one example of this, you might consider adding this feature by doubling the horizontal velocity of the ball the seventh time it hits the paddle, figuring that's the time the player is growing complacent.

- *Keep score.* You could easily keep score, generating points for each brick. In the arcade game, bricks were more valuable higher up in the array, so that you got more points for red bricks than cyan bricks. You could display the score underneath the paddle, since it won't get in the way there.

- *Use your imagination.* What else have you always wanted a game like this to do?

Grading

Functionality: Your code should compile without any errors or warnings. We will run your program with a variety of different **constant** values to test whether you have consistently used constants throughout your program.

In general, for the required parts of the assignment, limit yourself to using Java syntax taught in lecture and the parts of the textbook we have read through July 11.

Style: A particular point of emphasis for style grading on this assignment is the proper usage of private instance variables. You should minimize the instance variables in your program; do NOT make a value into an instance variable unless absolutely necessary. Write a brief comment on each instance variable in your code to explain what it is for and why you feel it necessary. All instance variables must be private.

Beyond this, follow style guidelines taught in class and listed in the course Style Guide. For example, use descriptive names for variables and methods. Format your code using indentation and whitespace. Avoid redundancy using methods, loops, and factoring. Use descriptive comments, including at the top of each .java file, atop each method, inline on complex sections of code, and a citation of all sources you used to help write your program.

Decomposition: Break down the problem into coherent methods, both to capture redundant code and also to organize the code structure. Each method should perform a single clear, coherent task. No one method should do too large a share of the overall work. Your `run` method should represent a concise summary of the overall program, calling other methods to do the work of solving the problem, but `run` itself should not directly do much of the work. In particular, `run` should *never directly create graphical components like the paddle, ball, or bricks*. Nor should it directly check for collisions or respond to them. Instead, you should delegate these tasks to other methods that are called by `run`.

Honor Code: Follow the Honor Code when working on this assignment. Submit your own work and do not look at others' solutions (outside of your pair, if you are part of a pair). Do not give out your solution. Do not search online for solutions. Do not place a solution to this assignment on a public web site or forum. Solutions from this quarter, past quarters, and any solutions found online, will be electronically compared. If you need help on the assignment, please feel free to ask.

Good luck, and have fun!