Nick Troccoli
CS 106A

Assignment 4
July 20, 2017

Assignment #4 — Artistry and Breakout
Due: 11AM PST on Monday, July 31st

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

Based on handouts by Marty Stepp, Mehran Sahami and Eric Roberts

The purpose of this assignment is to practice creating graphical programs, using concepts such as events, animation and instance variables. There are two parts; the first, **Artistry**, implemented in `Artistry.java` (and `Artistry2.java`, if you are in a pair) lets you dream up and create your own custom graphics. The second, **Breakout**, implemented in `Breakout.java`, is an animated arcade game that is 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 release of this assignment (July 20). 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.

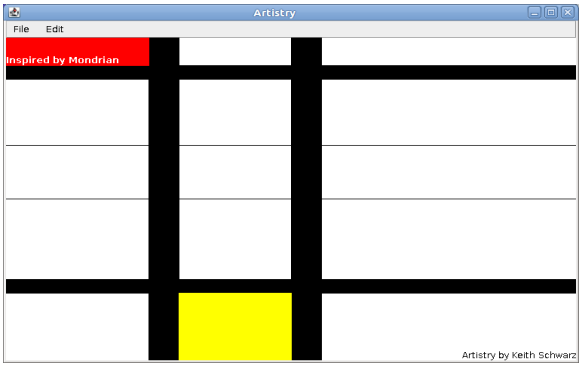
Part 1: Artistry

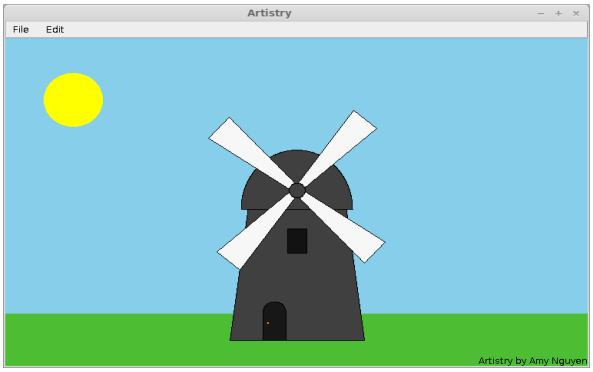
For this part of the assignment, turn in a `GraphicsProgram` named **Artistry** that draws any graphical figure of your choice, with the following constraints:

- Its size must be at least 100x100 pixels
- It must contain at least three different kinds of shapes (e.g. `GRect`, `GOval`, `GLine`, etc.)
- It must use at least two different colors other than black and white
- It must be your own work, and it must not be highly similar to the other graphical figures you draw elsewhere in this assignment or ones that were shown in class or the textbook. (This is subjective, but the idea is, you should come up with your own drawing and not simply turn in one of ours or a trivially modified version thereof).
- Your program should not have any infinite loops and should not have any user interaction
- You must "sign your name" in the bottom-right corner. To do this, create a `GLabel` with the text "*Artistry by NAME*", where NAME is your name. (This `GLabel` doesn't count as one of the three different types of `GObjects` that you're required to have.) If you work in a pair, each student should do their own Artistry, one as `Artistry.java` and the other as `Artistry2.java`. Make sure to comment and sign your name on each.

Align the label so that it is flush up against the bottom-right corner of the window. Be sure that all the text is visible and that none of the letters in the **`GLabel`** are cut off. To be specific, the "descent" of letters such as "y" (the lower-hanging part of the letters) should be visible on the screen and not cut off. You should use the **`getDescent()`** method on the **`GLabel`** to find out the number of pixels that such letter descents occupy and adjust the **`GLabel`**'s onscreen location accordingly.

Your score for this problem will be based solely on functionality as just defined; it is not graded on style. The goal here is to let you practice graphics and play around a bit while allowing you to be creative and make something neat of your own creation. Be creative! As inspiration, here are a few Artistry figures drawn by past students:





Part 2: Breakout

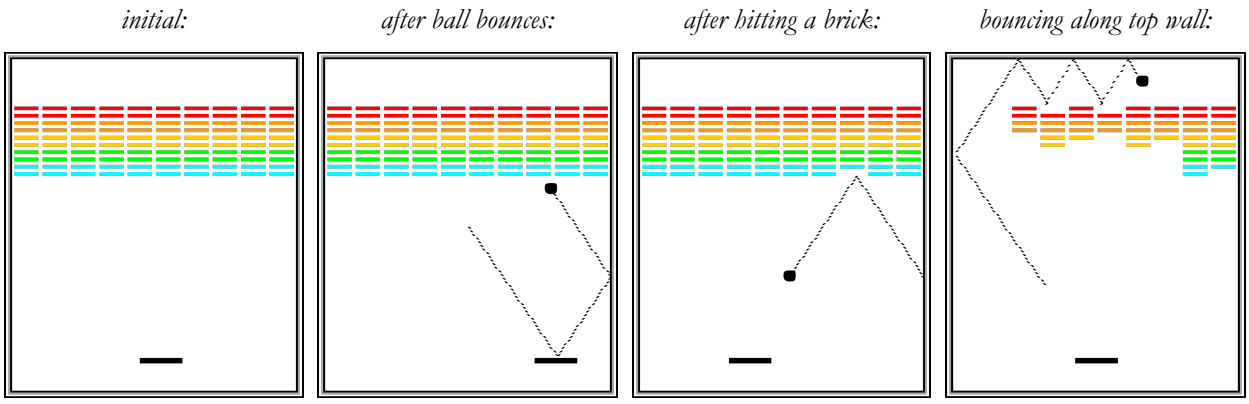
This part of the assignment will be the bulk of your work. Your job specifically is to write the classic arcade game of Breakout, which was invented by Steve Wozniak before he founded Apple with Steve Jobs. It is a large assignment, but entirely manageable as long as you follow the following pieces of advice:

- *Start as soon as possible.* This assignment is due in just over a week-and-a-half, 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.

In the starter project, we provide you a base file **`BreakoutProgram.java`** that your **`Breakout.java`** file extends, which includes several constants you must use in your code. These constants control the game parameters, such as the dimensions of the various objects. Your code should use these constants so that, if one changes, your program behavior adjusts accordingly. Each of the following sections lists relevant constants you should use in that section. You are welcome to add more constants, but *please do so in* **`Breakout.java`**, *not* **`BreakoutProgram.java`**.

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.

A fully-functioning Breakout demo program is available on the course website; if you have any questions about the game behavior, try out the demo program.

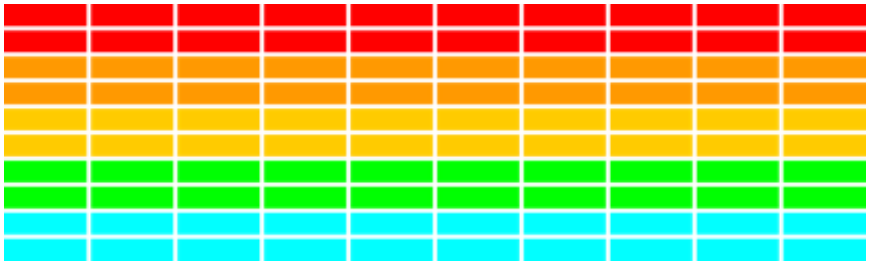
Approach

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 and End of Game

Stage 1: Bricks

Our first suggested task is to create the rows of bricks at the top of the game, which look like this:



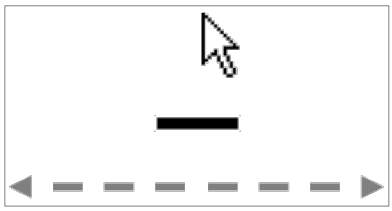
The number, dimensions, and spacing of the bricks are specified using constants in **`BreakoutProgram.java`**. Each brick should be a filled colored rectangle of size **`BRICK_WIDTH`** by **`BRICK_HEIGHT`**, with the top row starting at a y-coordinate of **`BRICK_Y_OFFSET`** (note that this offset is the distance from the top of the screen to the *top* of the first row). There are **`NBRICK_ROWS`** total rows, with **`NBRICK_COLUMNS`** bricks in each row. There is a gap of **`BRICK_SEP`** pixels between neighboring bricks in both dimensions. You need to compute the x coordinate of the first column 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 fewer than 10 rows. Your code should work for any reasonable number of rows. (If there are more than 10 rows, "wrap around" to make rows 11-12 red, 13-14 orange, 15-16 yellow, etc.)

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.

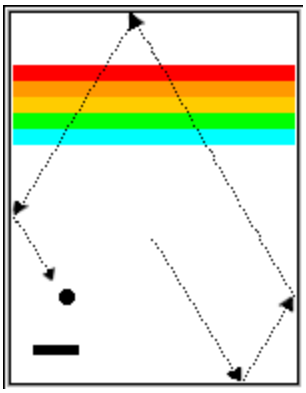


The paddle horizontally follows the mouse as it moves onscreen; specifically, you must make the horizontal center of the paddle follow the mouse. Mouse tracking uses events discussed in Chapter 10 of the textbook, and in lecture on 7/19 and 7/20. You only have to pay attention to the *x* coordinate of 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).



Create a filled black **`GOval`** and put it in the center of the window. (Remember that the coordinates of a **`GOval`** represent its upper left corner, not its center!). Now, let's get the ball to move properly. The program needs to keep track of the velocity of the ball, which consists of two separate components, one for the x dimension and one for y. You may create two private instance variables for these, as they will be used throughout your program and are useful "game state".

The velocity components represent the change in ball position on each time step of the animation. Initially, the ball should head downward with a velocity of **`VELOCITY_Y`**. (Recall that y values in Java increase as you move down the screen.) The game would be boring if every ball took the same course, though, so you should choose the x component of the velocity randomly; you can use a RandomGenerator to do this. Set your x velocity to be a random real number between **`VELOCITY_X_MIN`** and **`VELOCITY_X_MAX`**, randomly in the + (right) or - (left) direction with equal probability. *Make sure to exclude the range* **`-VELOCITY_X_MIN`** *through* **`VELOCITY_X_MIN`**; those lead to a ball going mostly straight down, which makes things too easy ☺.

Now the ball must bounce off the edges of the game world (ignoring the paddle and bricks for now). You can check to see if the ball's coordinates have gone beyond the world boundary, taking

into account that the ball's size. (Use **`getWidth()`** and **`getHeight()`** to find the game world's size.) To see if the ball has bounced off the wall, check whether any of the edges of the ball have become less than 0 or greater than the width/height of the canvas. For now, just have the ball bounce off the bottom wall, rather than worrying about ending the player's turn, so that you can watch it make its path around the world (we'll change this later). Thus, 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`

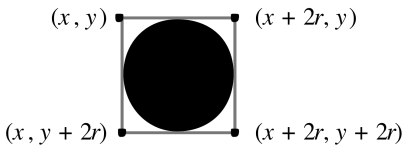
Stage 4: Collisions

Now, in order to make Breakout into a real game, you have to detect when the ball collides with another object in the window. If you look in Chapter 9 (page 299) at the methods defined in **`GraphicsProgram`**, you will see that there is a method **`getElementAt(double x, double y)`** that takes x and y coordinates in the window as parameters and returns the **`GObject`** at that location, if any. If there are no graphical objects that cover that position, **`getElementAt`** returns the special value **`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 canvas (the one added most recently).

By calling **`getElementAt(x, y)`**, where **`x`** and **`y`** are the coordinates of the ball, if the point **`(x, y)`** is contained within 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; any part of the ball might collide with something on the screen. 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. If you find something at one of those points, you know that the ball has collided with that object.

In your implementation, you should check the four corner points on the bounding square around the ball. Remember that a **`GOval`** is defined in terms of its bounding rectangle, so that if the upper left corner of the ball is at (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 are 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.

When writing this functionality, you should work to reduce redundancy and have clean code. It might be a good idea to write the above code as its own **method** such as

```
private GObject getCollidingObject()
```

that returns the object involved in the collision, if any, and `null` otherwise. You could then store the value that is returned into a variable (called, for instance, `collider`) by saying

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

You must then decide what to do in your code when a collision occurs, based on what the ball collides with; there are only two possibilities. First, the object you get back might be the paddle, which, if you stored the paddle in an instance variable, 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. You need to cause a bounce in the vertical direction, but you also need to take the brick away. To do so, remove it from the screen by calling the `remove` method and passing that brick as a parameter.

Relevant constants: `None`

Polish: At this point, you've completed most of the difficult parts of the assignment. Congratulations! Now we recommend that you test your program thoroughly by playing it for a while. In particular, try temporarily changing our pre-defined constants and making sure that your code adapts properly and still works. 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`.)

Stage 5: Turns and End of Game

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

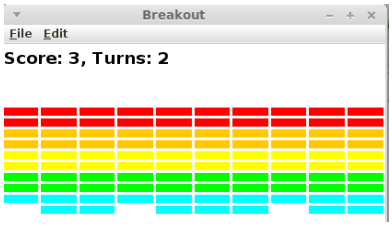
Turns and end of turn: The player initially has 3 turns (constant: `NTURNS`) remaining. Every time the ball hits the bottom edge of the window, the player loses 1 turn. When the turn ends, if the player has more turns remaining, your program should re-launch the ball from the center of the window toward 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.

Game info label: You must add a `GLabel` to your canvas that displays the player's current score and number of turns remaining. Initially, the player has a score of 0 and has 3 turns (constant: `NTURNS`) remaining. The label's text should be of the format `"Score: __, Turns: __"` and

should use the font string constant **`SCREEN_FONT`** as its font. As a reminder, to make your label use this font, write a line such as the following:

```
myLabel.setFont(SCREEN_FONT);
```

The label should be located at the top/left corner of the window. Note that, because a label is positioned according to the *far-left of its baseline*, this is not (0, 0), it is (0, *label height*). Also note that, because this is another onscreen graphical object (besides the bricks, paddle and ball), you will need to update your collision logic to make sure that the ball *does not bounce* off of this label as though it is a brick. No collisions should occur between the label and the ball.



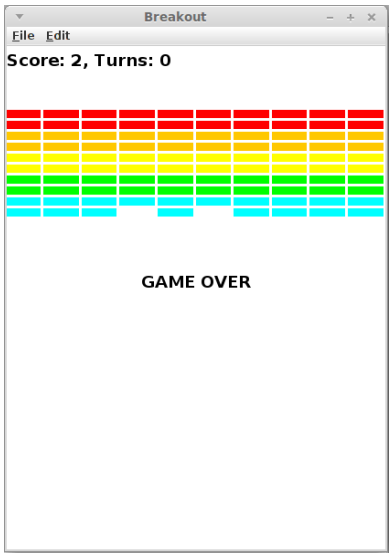
Every time the player hits a brick with the ball, they score **1 point**. The label should immediately update to reflect this. Every time the ball hits the bottom edge of the window, the player loses **1 turn**. The label should immediately update to reflect this.

Winning the game: If the player successfully removes all bricks from the screen (before they run out of turns), they win! The easiest way to check for this condition is to keep a count of the number of bricks remaining, and decrease it by one every time a brick is removed. If the count reaches zero, the player has won. There are a few actions your program should take when this happens (see screenshot at right):

1) You should make sure your **game info label** displays the correct score (accounting for all bricks being removed)
2) You should show a centered **`GLabel`** saying `"YOU WIN!"`.
3) You game must not freeze, crash, or throw an exception when you reach the end of the game.

Losing the game: If the player loses their last turn, the game ends. There are a few actions your program should take when this happens (see screenshot at right):

1) You should update your **game info label** to indicate that the player has 0 turns remaining.
2) You should hide or remove the ball and paddle from the screen so that they are not visible.



Relevant constants: `NTURNS`, `SCREEN_FONT`

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:

- **Sounds:** Add sound effects, such as a sound for the ball bouncing off of something. The starter project contains an audio file called `bounce.au` in the `res/` subdirectory, but feel free to add your own as well. You can load and play a sound by writing:

  ```
  AudioClip bounceClip = MediaTools.loadAudioClip("res/bounce.au");
  hornClip.play();
  ```

  Note that for this you will need to add `import acm.util.*` and `import java.applet.*` to the top of your .java file.

- **Additional labels and pauses:** Optionally, before each round, you can make the program wait for the user to click the mouse by calling the `waitForClick()` method, which causes your program to pause until the user clicks the mouse once. Once the user clicks, serve the ball to begin a turn. You can also show a text label saying, "Click to begin", etc., or other labels throughout the game to make the game easier to understand for the user.

- **Improved bouncing:** Improve the ball control when it hits different parts of the paddle. Make the ball bounce in both the x and y directions if you hit it on the edge of the paddle from which the ball was coming.

- **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 doubling the horizontal velocity of the ball the seventh or so time it hits the paddle, figuring that's the time the player is growing complacent.

- **Improved score:** In the arcade game, bricks in higher rows were worth more points.

- **Power-ups**: Add power-ups (or penalties!) that the user gets when hitting certain bricks. For instance, one brick could contain a "paddle expand" power-up, another could add a second ball to the screen, and a third could contain a "paddle shrink" penalty.

- **Other games:** There are other games that are very similar to breakout, such as Pong. Can you write one?

- **Other:** Use your imagination! What other features can you imagine in a game like this?

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 20.

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. For full credit, 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.

Copyright © Stanford University and Nick Troccoli, licensed under Creative Commons Attribution 2.5 License. All rights reserved.