Stanford University, CS 106A, Homework Assignment 4
Breakout (PAIR ASSIGNMENT)

Based on past versions of this assignment created by Eric Roberts, Mehran Sahami, Keith Schwarz, and others.

The purpose of this assignment is to practice creating graphical programs, along with new concepts such as fields and events. The main task in this assignment will be to create an animated game called **Breakout**. There is a **starter project ZIP archive** on the class web site. You will **turn in** your project with:

- **Artistry.java**, your implementation of the Artistry exercise for Part A *(* also **Artistry2.java** if in a pair)*
- **Breakout.java**, your implementation of the Breakout game for Part B

This is a **pair assignment**. You may work in a pair, or you may work individually. 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. Submit using the Submit Project entry in the Stanford Menu.

Part A: Artistry

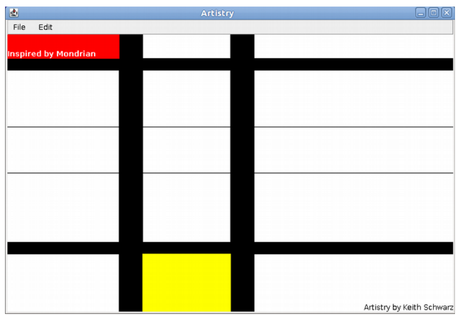
For this part of the assignment, turn in a `GraphicsProgram` named **`Artistry`** that draws any graphical figure of your choice. You may draw *any figure you like*, as long as you meet the following constraints:

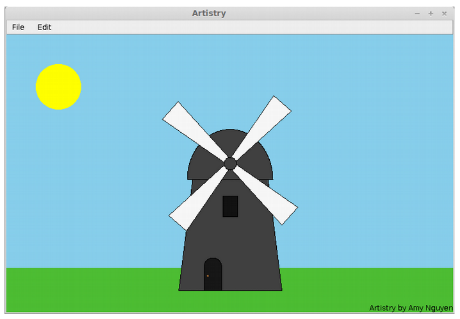
- Its size must be at least 100 x 100 pixels.
- It must contain at least three different kinds of shapes (e.g. `GRect`, `GOval`, `GLine`, etc.).
- It must use at least two different colors other than black and white
- It must be your own work. (If you work in a **pair**, each student should do their own Artistry, one as **Artistry.java** and the other as **Artistry2.java**. Make sure to comment and sign your name on each.)
- It must not be highly similar to the other graphical figures you draw on other problems of 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 read any user input (e.g. `readInt`).
- 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 (or both of your names, if you are working in a pair). For example, a pair would include a label such as, "Artistry by John Smith and Jane Doe". (This `GLabel` doesn't count as one of the three different types of `GObjects` that you're required to have.)

  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, rather than having every problem on this assignment be rigidly specified.

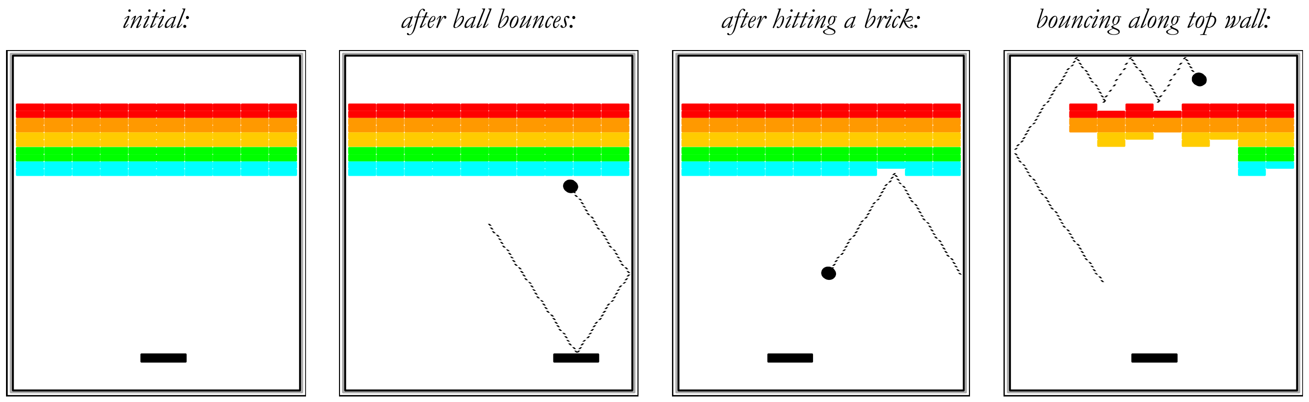
Be creative! As inspiration, here are a few Artistry figures drawn by past students.





Part B: Breakout!

This Part will be the bulk of your work and is the main exercise for this assignment. The rest of this spec discusses Breakout in detail. The rest of this spec should be assumed to apply to Breakout only and not to Artistry.



The classic arcade game of **Breakout** was invented by Steve Wozniak before he founded Apple with Steve Jobs. In this game, 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 until it reaches the edge of its space. 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 figure above shows the ball's path after two bounces, one off the paddle and one off the right wall. *(The dotted line in our diagrams is there to show the ball's path and won't actually appear on the screen.)*

When the ball collides with a brick or wall, the ball bounces back and the brick disappears. The third figure above shows what the game looks like after that collision. The player is moving the paddle leftward so the ball will hit it.

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

1. The ball hits the lower wall, which means that the player 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 game ends in a loss.
2. The last brick is eliminated. In this case, the player wins, and the game ends immediately.

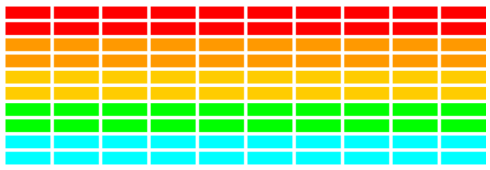
After all the bricks in a particular column have been cleared, a gap will open to the top wall. Now it becomes possible for the ball bounce several times between the top wall and the upper line of bricks. This condition is called "breaking out," as shown in the far-right diagram above. Even though "breaking out" is an exciting part of the game, you don't have to do anything special in your code 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.

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 stage is described in more detail on the following pages:

1. Bricks
2. Paddle
3. Ball
4. Collisions
5. Turns and end of game

To make our implementation more flexible, we will provide you a base file **BreakoutProgram.java** that you must extend, which includes several constants you must use in your code. If a constant changes, your program must adjust its behavior accordingly.

Stage 1: Bricks

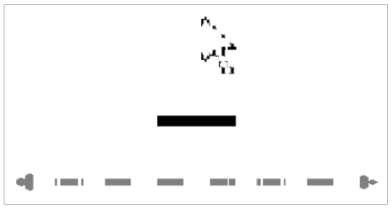


Our first suggested task is to create the rows of bricks at the top of the game, which look like the figure at right. The number, dimensions, and spacing of the bricks are specified using named **constants** in the starter file, as is the distance from the top of the window to the first line of bricks. 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`. 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

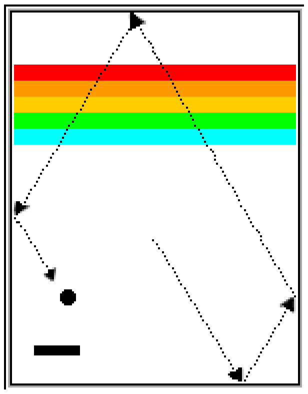


Our next suggested task is to create the paddle. In a sense, this is easier than the bricks. There is only one paddle, which is a filled black `GRect`. You must set its size and *y*-position relative to the bottom of the window via **constants**.

You must make the horizontal <u>center</u> of the paddle follow the mouse. Mouse tracking uses events from textbook Ch.9. You only need to monitor the *x*-coordinate of the mouse; the paddle's *y* position is fixed. Do not allow any part of the paddle to move off the edge of the window. Check to see whether the *x*-coordinate of the mouse extends beyond the boundary and ensure that the <u>entire</u> paddle is visible in the window.

- *Relevant constants:* `PADDLE_WIDTH`, `PADDLE_HEIGHT`, `PADDLE_Y_OFFSET`

Stage 3: Ball (and bouncing)



Now let's make the ball. Create a filled black `GOval` and put it in the center of the window. (The coordinates of a `GOval` represent its upper left corner, not its center.)

Next, 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 **fields** (private instance variables) for these.

The velocity components represent the change in position that occurs on each time step of the animation. Initially, the ball should be heading downward, and you might try a starting velocity of positive `3.0` for your *y* velocity. (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, so you should choose the *x* component of the velocity **randomly**.

Use a `RandomGenerator` object to get random numbers. Set your *x* velocity to be a random real number between `1.0` and `3.0`, randomly in the `+` (right) or `-` (left) direction with equal probability. Make sure to exclude the range `-1.0` through `1.0`; those lead to a ball going more or less straight down, which is too easy for the player.

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 `getCanvasWidth()` and `getCanvasHeight()` to find the game world's size.) To see if the ball has bounced off the wall, check whether the coordinate of the edge of the ball has become less than 0 or greater than the width/height of the canvas. When the ball bounces off a wall, **reverse the sign of the appropriate velocity component, `vx` or `vy`**. 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 that test later so that hitting the bottom wall signifies the end of a turn.

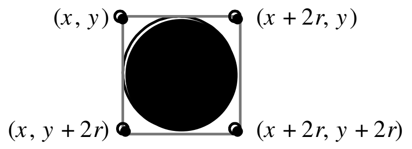
- *Relevant constants:* `BALL_RADIUS`, `VELOCITY_MIN`, `VELOCITY_MAX`, `DELAY`

Stage 4: Collisions

In order to make Breakout into a real game, you have to detect when the ball collides with another object in the window. Textbook Chapter 9 (page 299) lists the methods that are defined in the `GraphicsProgram`. There is a pre-existing method named `getElementAt` that will be useful. You can call this method, passing *x*/*y* parameters for a pixel position in the window, and it will return the graphical object found at that location, if any. If there are no graphical objects that cover that position, it returns the special constant value `null`. If there is more than one element found at the same location, `getElementAt` always gives you the one that appears to be in front on the display's z-ordering, which corresponds to the one that was added to the window last. By calling `getElementAt` and passing the coordinates of the ball, if the given (*x*, *y*) point is underneath an object, this call returns the graphical object with which the ball has collided. If the method returns the value `null`, there is no collision.

But the ball is not a single point; any part of the ball might collide with something on the screen. What we'll do is check a few chosen points on the outside of the ball and see whether any of those points has collided with anything. If you find a non-`null` value at any one of those points, you know that the ball has collided with an object.

In your code you'll check the four corner points on the bounding square around the ball. If the top-left corner of the ball's `GOval` is at (*x*, *y*), the other corners will be at the locations in the diagram at right.



These four points are outside the ball, so `getElementAt` won't return the ball itself, but the points are close enough to make it appear that collisions have occurred. **For each of the four points** shown, you need to:

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

A lot of students introduce redundant or clunky code here to handle the four possible collision points. We look for this in grading. You should work to reduce redundancy and have clean code for this part of your game. It might be a good idea write the above code as its own **method** that returns the object currently involved in a collision with the ball, if any, otherwise `null`. You could call it and store its returned result as a `GObject`, `GRect`, `GOval`, etc.

Another possibly simpler way that you can look for collisions is to pass multiple x/y pairs to `getElementAt`. The method accepts any number of x/y pairs as parameters and will check each one in order and return the object found at first x/y pixel that contains any object, or `null` if none of the x/y pairs contain an object. For example, if you had four x/y points represented as (x1, y1), (x2, y2), (x3, y3), and (x4, y4), you could write the following code:

```
// check multiple (x, y) locations in a single call
GObject collider = getElementAt(x1, y1, x2, y2, x3, y3, x4, y4);
```

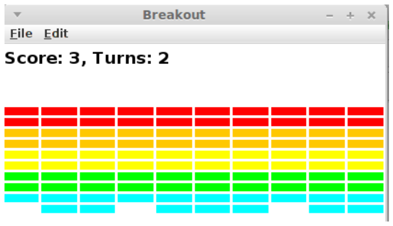
Your code must decide what to do when a collision occurs, based on what the ball collides with. If the ball collides with the paddle, change your program's state so that the ball will start traveling upward. (If you've stored the paddle as a field and the current collider as a variable, you can perform a test such as, `if (collider == paddle) {...` to check for this.) If it isn't the paddle, it must be a brick, since those are the only other objects in the world. Make the ball bounce vertically, but also clear the brick away by calling the `remove` method on your graphical program and passing that brick as a parameter.

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, and 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

If you've gotten here, you've almost done! There are a few more details you need to take into account:

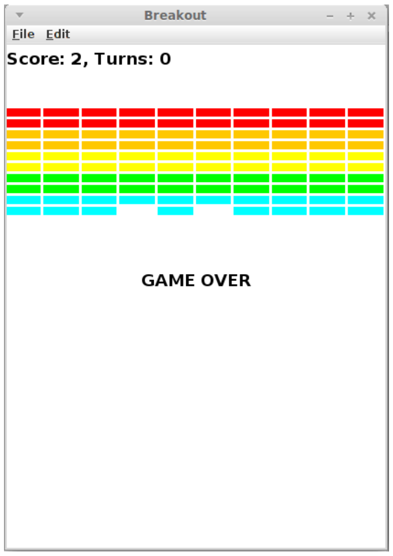
Turns and end of turn: The player initially has 3 turns (constant: `NTURNS`) remaining. Every time the ball hits the bottom edge of the window, the player loses **1 turn**. When the turn ends, if the player has more turns remaining, your program should re-launch the ball from the center of the window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



Score: 3, Turns: 2

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. If you like, you may optionally show a text label saying, "Click to begin" etc. Waiting for a click on each turn is <u>not</u> required.

- *Relevant constants:* `NTURNS`



Score: 2, Turns: 0

GAME OVER

Score and turns label: You must add a `GLabel` to your screen 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: __"`.

This label should use a 16-point bold **font**. We supply such a font for you as a constant called `SCREEN_FONT`. To make your label use this font, write a line such as the following:

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

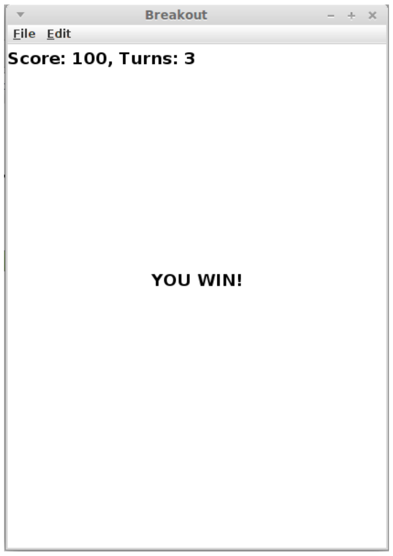
The label should be located at the top/left corner of the window. Note that this is not (0, 0). It is (0, *label height*).

Every time the player hits a brick, they score **1 point**. The score label should update immediately.

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

A common **bug** here is that the game ball will bounce off of the score label as though it were a brick. You must avoid this incorrect behavior. No collisions should occur between your score/turns label and the ball.

- *Relevant constants:* `SCREEN_FONT`



Score: 100, Turns: 3

YOU WIN!

Losing the game: When the player loses their last turn, the game ends. There are a few actions your program should take when this happens. You should update your **score/turns label** to indicate that the player has 0 turns remaining. You should also **hide** or remove the ball and paddle from the screen so that they are not visible. See screenshot at right.

Winning the game: Now you'll need to check for the other terminating condition: hitting the last brick. The easiest way to check for this is to keep a count of the number of bricks remaining. Every time you hit one, decrease your counter by one. When the count reaches zero, the game is done.

When the player clears the last brick, you must pop up a centered `GLabel` saying, **"YOU WIN!"** as shown at right. Your game must not "lock up," crash, or throw an exception when the end of game is reached.

Extra Features

Extra features are optional and will earn you a small amount of extra credit.

There are many possibilities for extra features that you can add if you like. If you are going to do extra features, **submit two versions** of the assignment: the basic **Breakout.java** that meets all the assignment requirements, and a **BreakoutExtra.java** containing your extended version. At the top of your **BreakoutExtra.java** file in your comment header, you must **comment** what extra features you completed. Here are a few ideas for extra features:

- *Sounds:* Make the ball make a sound when it bounces off of something. The starter project contains an audio clip file called **bounce.au** in its **res/** subdirectory. You can load and play a sound by writing:
```
AudioClip bounceClip = MediaTools.loadAudioClip("res/bounce.au");
bounceClip.play();
```
- *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.
- *Improved score:* In the arcade game, bricks in higher rows were worth more points.
- *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 could you imagine in a game like this?

Grading

Functionality: Your code should compile without any errors or warnings. When we run your game, the bricks, paddle, and ball should appear onscreen at the proper places, sizes, shapes, colors, etc. During the game, animation and collisions with objects such as walls, paddle, and bricks should alter the ball's position and velocity properly, and bricks should disappear as they are hit. Collisions with the bottom wall should end a turn. The overall game flow should work, where the game ends after 3 turns or after all bricks are cleared. We will run your program with a variety of different **constant** values to test whether you have consistently used constants throughout your program.

Style: A particular point of emphasis for style grading on this assignment is the proper usage of **fields** (a.k.a. private instance variables). You should minimize the fields in your program; do NOT make a value into a field unless absolutely necessary. Write a brief **comment** on each field in your code to explain what it is for and why you feel it necessary to make that into a field. All fields must be private. *(For reference, our solution to this assignment has 7 fields, 3 of which are graphical objects and 4 of which are numbers; but you don't need to match thins exactly. It is merely a rough guide.)*

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 the top of each .java file, atop each method, inline on complex sections of code, next to each field, 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. As a rough estimate, a method whose body has more than 30 lines is probably too large. Your `run` function 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`.

In general, limit yourself to using Java syntax taught in lecture and the parts of the textbook we have read so far.

Honor Code: Follow the **Honor Code** when working on this assignment. Submit your own work and do not look at others' solutions. Do not give out your solution. 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.