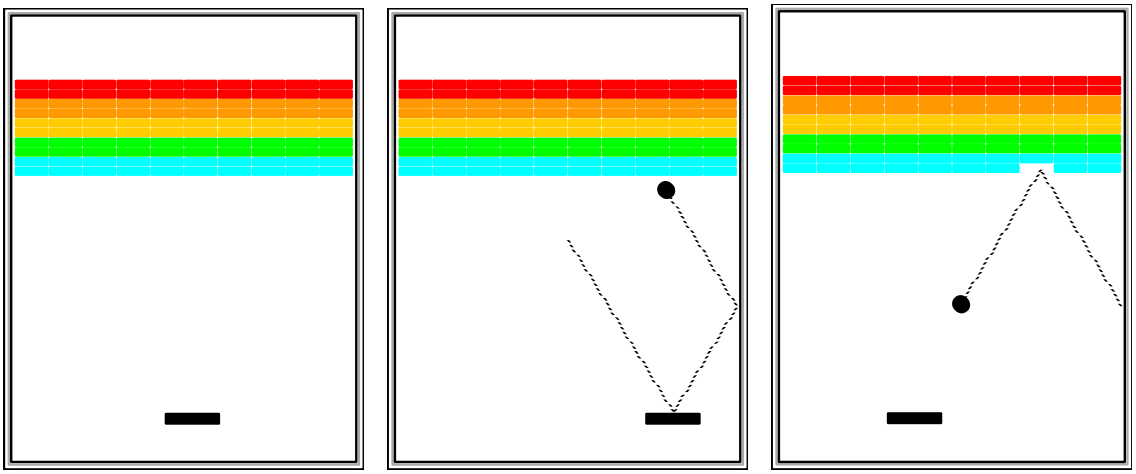
Project: Breakout!

A classic CS106A assignment developed by Eric Roberts and Mehran Sahami

As a final project for the Girl Code workshop, we recommend trying to make the classic arcade game *Breakout!* This project is a time-honored tradition here in Stanford's introductory CS classes and is a great way to pull together all the skills you've acquired so far. Plus, once you're done, you'll have a *very* nifty piece of software!



The Breakout Game

In Breakout, the initial configuration of the world appears as shown on the left. The colored rectangles in the top part of the screen are bricks, and the slightly larger rectangle at the bottom is the paddle. The paddle is in a fixed position in the vertical dimension, but moves back and forth across the screen along with the mouse until it reaches the edge of its space.

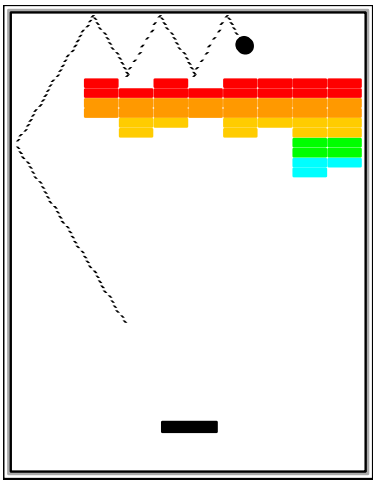
A complete game consists of three turns. On each turn, a ball is launched from the center of the window toward the bottom of the screen at a random angle. That ball bounces off the paddle and the walls of the world, in accordance with the physical principle generally expressed as "the angle of incidence equals the angle of reflection" (which turns out to be very easy to implement as discussed later in this handout). Thus, after two bounces—one off the paddle and one off the right wall—the ball might have the trajectory shown in the second diagram. (Note that the dotted line is there to show the ball's path and won't appear on the screen.)

As you can see from the second diagram, the ball is about to collide with one of the bricks on the bottom row. When that happens, the ball bounces just as it does on any other collision, but the brick disappears. The third diagram shows what the game looks like after that collision and after the player has moved the paddle to put it in line with the oncoming ball.

The play on a turn continues in this way until 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 game ends in a loss for the player.
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" and gives meaning to the name of the game. The diagram on the right shows the situation shortly after the first ball has broken through the wall. 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 file for this project 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 last page of this handout). 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.

This project might look complex, but it can be nicely broken down into a smaller series of tasks, each of which requiring not all that much code. The next few sections describe a reasonable staged approach to the problem. While you're free to write Breakout however you'd like, we suggest using our decomposition, since each piece nicely builds off of the previous pieces.

Step One: Create the Paddle

The first step is to create the paddle. The paddle should be controlled by the mouse so that whenever the player moves the paddle left or right, the paddle repositions itself horizontally so that it is centered underneath the mouse cursor. The paddle always sits at the same Y coordinate and never moves up and down.

When the player moves the mouse far to the sides of the window, you should ensure that the paddle doesn't end up partially off-screen. One of the earlier lab problems (the one with the cat chasing the mouse) asked you to solve this problem, so if you haven't worked through that problem yet, you might want to do so before moving on.

Step Two: Create a Ball and Get it to Bounce off the Walls

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. To start, create a ball and put it in the center of the window. You've centered objects before, and this step isn't any different than that.

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;
```

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 from last week, you should initialize **vx** as follows:

1. Declare a local variable of type **RandomGenerator** by writing

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

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

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

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

Once you've done that, your next challenge is to get the ball to bounce around the world, completely ignoring the paddle (the ball should pass right through it for now). 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. 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**.

In the bouncing ball demo we did earlier, you needed to worry about the case where the ball gets stuck in the walls or the floor. You actually don't need to worry about this here. In the bouncing ball case, if the ball entered the floor, it might not have enough energy to get back out on the next frame because the ball slows down on collisions. In *Breakout!*, collisions don't cause the ball to lose any energy, so there's no need to push the ball out of the walls or ceiling on impact.

Step Three: Handle Paddle Collisions

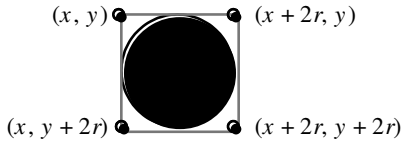
Right now, the ball bounces around the window, but completely ignores the paddle. That's probably not a good thing, so in this step you'll get the ball to bounce off of the paddle!

You might remember the helpful **getElementAt** method. If you call **getElementAt** and provide an *x* and *y* coordinate, you'll get back a value (of type **GObject**) representing the object that was hit. If it's some object, you'll get back that object. If you didn't hit anything at all, you'll get back the special value **null**. (Check out the Debris Sweeper demo for an example of this).

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) . . .
```

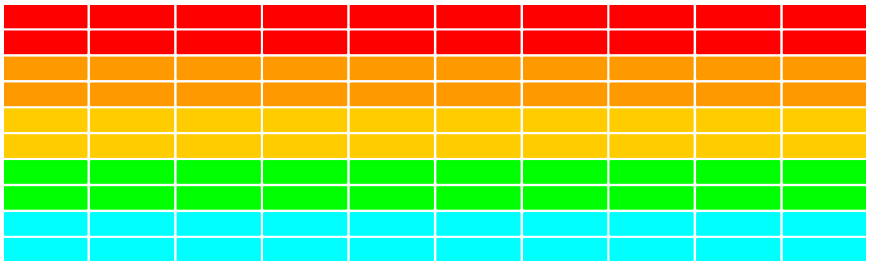
If it is the paddle, you need to bounce the ball so that it starts traveling up. If it isn't the paddle, then (at this point in development) it must be **`null`** and you didn't hit anything at all.

Right now, there's no way to win or lose the game. The ball bounces off the bottom wall just like any other wall. Now, change this so that when the ball drops below the bottom of the screen, the player loses the game.

Step Four: Add Bricks

Right now, you have a different game called *Pong* in which you need to bounce the ball without it falling below the screen. To convert this to *Breakout!*, you just need to add bricks.

Change your code so that it begins by drawing a 2D grid of bricks on the top of the screen:



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. The color of the bricks remain constant for two rows and run in the following rainbow-like sequence: **`RED`**, **`ORANGE`**, **`YELLOW`**, **`GREEN`**, **`CYAN`**.

You've already drawn 2D grids of objects with space between them (remember the optical illusion example?), so we hope this won't be too tough! For now, don't worry about having the ball bounce off the bricks – that's the very last step!

Step Five: Handle Brick Collisions

You're almost there! Right now, the only thing missing is getting the ball to break the bricks.

Currently, your code checks for collisions by seeing whether the ball hit the paddle. There are only two possible options for what happens when you call **`getElementAt`** – either you hit the paddle, or you hit nothing. Now, there are *three* options – you either hit the paddle, or you hit nothing, or you hit something other than the paddle. In that last case, you must have hit a brick – there's nothing else on the screen that you could have possibly hit!

Change your code for collision detection so that if the ball hits an object other than the paddle, you remove that object (using the **`remove`** method; see the Debris Sweeper example for details) and change the ball's y velocity as if it hit the top of the screen or the paddle.

Step Six: Final Touches!

Home stretch! All that's left to do now is some minor cleanup and changes. Each of these steps should be relatively straightforward:

- You've got to give the player three chances to break all the bricks. 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. Once the user clicks, you can serve the ball.
- 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. If you think everything is working, here is something to try: Just before the ball is going to pass the paddle level, move the paddle quickly so that the paddle collides with the ball rather than vice-versa. Does everything still work, or does your ball seem to get "glued" to the paddle? If you get this error, try to understand why it occurs and how you might fix it.

```
/*
 * File: Breakout.java
 * -------------------
 * This file will eventually implement the game of Breakout.
 */

import acm.graphics.*;
import acm.program.*;
import acm.util.*;
import java.awt.*;
import java.awt.event.*;

public class Breakout extends GraphicsProgram {

   /** Width and height of application window in pixels */
   public static final int APPLICATION_WIDTH = 400;
   public static final int APPLICATION_HEIGHT = 600;

   /** Dimensions of game board (usually the same) */
   private static final int WIDTH = APPLICATION_WIDTH;
   private static final int HEIGHT = APPLICATION_HEIGHT;

   /** Dimensions of the paddle */
   private static final int PADDLE_WIDTH = 60;
   private static final int PADDLE_HEIGHT = 10;

   /** Offset of the paddle up from the bottom */
   private static final int PADDLE_Y_OFFSET = 30;

   /** Number of bricks per row */
   private static final int NBRICKS_PER_ROW = 10;

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

   /** Separation between bricks */
   private static final int BRICK_SEP = 4;

   /** Width of a brick */
   private static final int BRICK_WIDTH =
     (WIDTH - (NBRICKS_PER_ROW - 1) * BRICK_SEP) / NBRICKS_PER_ROW;

   /** Height of a brick */
   private static final int BRICK_HEIGHT = 8;

   /** Radius of the ball in pixels */
   private static final int BALL_RADIUS = 10;

   /** Offset of the top brick row from the top */
   private static final int BRICK_Y_OFFSET = 70;

   /** Number of turns */
   private static final int NTURNS = 3;

   public void run() {
      /* You fill this in, along with any subsidiary methods */
   }
}
```