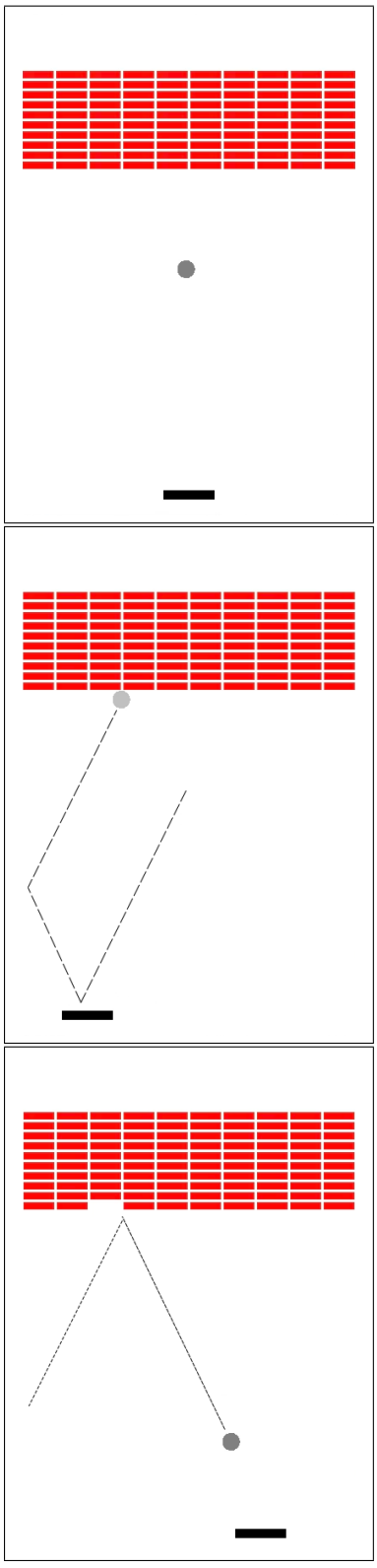
`Breakout.java`

1 Purpose

This assignment builds on your knowledge of the acm.graphics package, helps you practice branches and loops, and makes you take a large problem and break it down into smaller parts.



2 Program Description

Write the classic arcade game of Breakout. This is a large assignment, but well within your capabilities, as long as you break the problem up into manageable pieces. To help with this, the project is due in 2 parts. Read further for details.

3 The Breakout Game

In Breakout, the initial "world" (the space inside the app window) appears as shown in the diagrams. The red rectangles in the top part of the screen are bricks, and the slightly larger, black rectangle at the bottom is the paddle. As the mouse moves, the paddle moves back and forth only, not up and down. The paddle cannot move off the screen. Run the demo to get a sense of how the game works.

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, following the Law of Reflection, or "the angle of incidence equals the angle of reflection." This 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 only to show the ball's path and won't actually 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:

- 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.

- The last brick is eliminated. In this case, the player wins, and the game ends immediately.

You can decide how to show that the user wins or loses when one of those conditions occurs.

4 The Breakout.java starter file

The Breakout.java file takes care of the following details:

1. It includes the imports you will need in writing the game.
2. 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. You can also change these constants to help you test your program. See the section on "Testing Your Program" for more details.
3. It defines a variable for the paddle (it is defined for you because it needs to be defined in a particular place so that both the run() method and mouseMoved() methods can use it).
4. It includes a run() method that sets up the mouse listener. You write the rest.
5. It includes a mouseMoved() method that you will use to move the paddle as the mouse moves.
6. It includes a main() method that starts the application program and sets the window to the appropriate size. You should ignore the main() method and just write the implementation of run() and mouseMoved().

As in Project 1, you will need to add the acm.jar to your new Netbeans project. In Netbeans, right click "Libraries" and "Add JAR/Folder" to add acm.jar to your project.

5 Planning Your Implementation

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.

6 Part 1 (see Brightspace for the due date): World Setup, Paddle Movement, Ball Physics

We suggest that you implement these sections in order, testing each one before moving on to the next.

6.1 Set up the bricks

Before you start playing the game, you have to set up the various pieces. An important part of the setup is creating the rows of bricks at the top of the game. 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.

Test your code by setting NBRICKS_PER_ROW to a small number (such as 5). The rows should still be centered horizontally in the screen.

Test your code by setting NBRICK_ROWS and BRICK_WIDTH to a different number.

6.2 Create the paddle

There is only one paddle, which is a filled GRect. Its bottom relative to the bottom of the window is a named constant. You must calculate where its top left coordinate is. It should be placed horizontally in the middle of the screen to start. Once you have put the paddle on screen, you must make it track the mouse. Variables for the x coordinate of the mouse and the current y coordinate of the paddle have already been created for you in the mouseMoved() method; you only have to pay attention to the x coordinate of the mouse, because the y position of the paddle is fixed. You should not let the paddle move off the edge of the window. Thus, you'll have to check to see whether the x coordinate of the mouse would make the paddle extend beyond the boundary and change it if necessary to ensure that the entire paddle is visible in the window.

6.3 Create a ball and get it to bounce off the walls

The ball is a filled GOval. First, create the ball and put it in the center of the window. Keep in mind that the (x,y) coordinates of the GOval do not specify the location of the center of the ball but rather its upper left corner.

The program needs to keep track of the velocity of the ball, which consists of two separate components for the x and y velocities. Declare these as doubles, for example vx and vy. The velocity components represent the change in position that occurs on each time step. Initially, the ball should be heading downward, so 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. Please refer to the zyBooks chapter on Variables/Assignments for a review of random number generation. We will be generating a random double in the range 0.0 to 1.0, and then manipulating that so that vx is between 1.0 and 3.0 like this:

```
vx = randGen.nextDouble() * 3.0 + 1.0;
```

Make it negative half the time by adding the following code (assuming randGen is the name of your Random variable):

```
if (randGen.nextBoolean()) {
    vx = -vx;
}
```

This strategy works much better for Breakout than generating a random x between -3.0 and 3.0, which might generate a ball going straight down.

In the run() method, you will see a loop that looks like this:

```
while (!isDone) {
    // TODO: write any code that happens over and over as the game plays.
    // adjust the value of PAUSE_TIME to change the animation speed.
    pause(PAUSE_TIME);
}
```

This is where you should change the location of the ball using vx and vy. The current location of the ball can be found using the ball's getX() and getY() methods. Use setLocation() to change the ball's location. Use the paddle code from mouseMoved() as a guide.

Once you've done that, your next challenge is to get the ball to bounce around the world, ignoring 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. Remember that the (x,y) coordinates refer to the ball's top left corner! 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.

7 Part 2 (see Brightspace for the due date): Collision Detection and Game Play Logic

7.1 Checking for 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? The Breakout class is a GraphicsProgram and has a method

```
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.

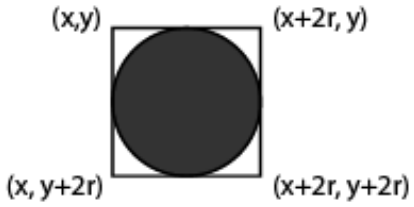
What happens if you call

```
GObject collision = 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 the 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 (other than the ball of course) 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. For example:

   ```
   GObject collider = getElementAt(x,y);
   ```

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.

From here, the only remaining thing you need to do is decide what to do with a collision, if it 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. For example:

```
remove(collider);
```

7.2 Game play logic

Here are the final details to implement:

1. 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 must modify your loop structure so that it tests for hitting the bottom wall as one of its terminating conditions.

2. 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.

3. Give the user 3 tries to clear the bricks. Please use a named constant for this.

4. 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?

8 Testing Your Program

Here are some suggestions for testing your program to make sure it works:

1. Play for a while and make sure that as many parts of it as you can check are working.

2. To quickly check whether you can win, set PADDLE_WIDTH to WIDTH (the entire width of the screen) and set PAUSE_TIME to 0.

3. Change the numbers in NBRICKS_PER_ROW and NBRICK_ROWS to make sure you win/lose the game in the same way with a different number of bricks.

4. With a normal paddle and pause time: 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.

9 Style Requirements for all Projects

- Use a multi-line comment at the top of your program that contains the name of your program, what project it is, your name, and the due date. See the Example.java file for an example.

- In your multi-line ("header") comment, please include a sentence pledging that you have upheld the Non-Collaboration Policy. (See the Syllabus for details on the policy).

- Use single line comments to describe what your code is doing. You don't have to comment every line.

- Variable names are camelCase, starting with lowercase
- Variable names are descriptive
- Variables are defined early (near the top of the main() method, where the others are already defined)
- Variables are given an initial, default, value when they are declared. For example: `int amount = -1;`
- Indentation is correct (use Netbeans to do this automatically, with Source → Format)

10 Grading Criteria: Part 1

Compiles with no syntax errors	10%
Header comment, program description, and pledged statement	5%
x coordinate of first brick column centers row of bricks in the world.	5%
loop correctly creates rows of bricks.	10%
named constants are used for brick, paddle, and ball creation.	10%
Ball starts in center, paddle starts in center at correctly calculated Y coordinate	10%
Ball initially moves downward at a random angle.	10%
Ball bounces off sides of the world using Law of Reflection	10%
Paddle stays on the screen at all times	5%
Paddle tracks mouse movement	5%
Variables have meaningful names and default values	5%
Code is indented correctly	5%
Named constants follow naming conventions and are used correctly	5%
Inline comments are used to explain sections of the code	5%

11 Grading Criteria: Part 2

Compiles with no syntax errors	10%
Collisions: brick removed when ball collides, ball bounces correctly.	20%
Collisions: ball bounces off paddle correctly.	20%
Game play: user gets 3 tries to clear the bricks.	15%
Named constant used for number of tries.	5%
Game play: user loses the try if the ball hits the bottom.	10%
Game play: user wins if all bricks are gone.	10%
UI: Something appropriate happens upon "game over" loss or win.	5%
Style: appropriate variable names, initial values, indentation, inline comments	5%

12 Reminder: Project Handin Rules

1. Projects may be turned in up to a week late. You will lose 5% of the grade for each day late.