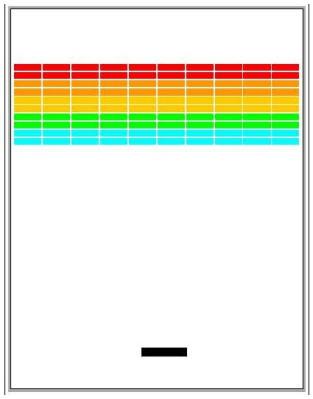


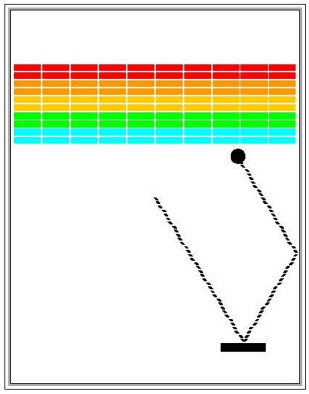
This assignment, including much of the wording of this document, is taken from an assignment from Stanford University, by Professor Eric Roberts.

Please keep track of the time you spent on this assignment. We will ask for it when it is time to submit.

Your task is to write the classic arcade game *Breakout*. The assignment will use the acm graphics package, which was used in A5. The assignment is easily within your grasp —as long as you break the problem up into manageable pieces and program and test incrementally. The decomposition is discussed in this handout, and, later, we give several suggestions for staying on top of the project .

Our solution to this assignment is about 320 lines long and has 11–12 methods (including `main` and `run`).

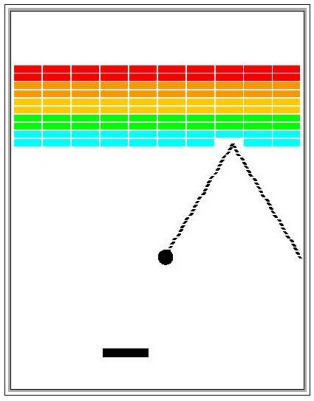
Breakout



The initial configuration of the game *Breakout* is shown above on the right. 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; it moves back and forth horizontally across the screen along with the mouse —unless the mouse goes past the edge of the window.

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. The 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" (it's easy to implement). The start of a possible trajectory, bouncing off the paddle and then off the right wall, is shown to the right. (The dotted line is there only to show the ball's path and won't actually appear on the screen.)

In the second diagram, the ball is about to collide with a brick 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 the game after that collision and after the player has moved the paddle to put it in line with the oncoming ball.

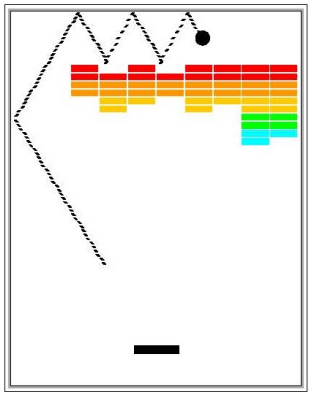


The play on a turn continues in this way until one of two conditions occurs:

1. The ball hits the lower wall, which means that the player missed it with the paddle. In this case, the turn ends. If the player has a turn left, the next ball is served; otherwise, the game ends in a loss for the player.

2. The last brick is eliminated: the player wins, and the game ends.

Clearing all the bricks in a particular column opens a path to the top wall. When this delightful 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 having to worry about hitting the ball with the paddle. This condition, a reward for "breaking out", gives meaning to the name of the game. The fourth diagram on the right shows the situation shortly after the first ball has broken through the wall. The ball goes on to clear several more bricks before it comes back down the open channel.



Breaking out is an exciting part of the player's experience, but 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.

Partners

You may work with one partner. Both of you should do what is necessary on the CMS to form a group several days before the submission deadline. It is dishonest and against academic integrity to split the work so that each does half of the work. You must work together the whole time.

File Breakout.java

We give you file Breakout.java, which is the only Java class that you will have to write. A partial skeleton of it appears to the right, to give you the basic idea. The class includes:

- The import statements needed in writing the game.
- 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.
- A static method `main` that starts the application program and sets the window to the appropriate size. Run the program from the Interactions window of DrJava by executing

```
Breakout.main(null);
```

```
import acm.graphics.*;
...
/** An instance is the game breakout. Start it by
    executing
        Breakout.main(null);  */
public class Breakout extends GraphicsProgram {
  /** Width of the game display  */
  private static final int WIDTH=  450;
  /** Height of the game display */
  private static final int HEIGHT=  610;

  /** Width of the paddle */
  private static final int PADDLE_WIDTH=  60;
  /** Height of the paddle */
  private static final int PADDLE_HEIGHT=  10;
  /** Offset of the paddle up from the bottom */
  private static final int
              PADDLE_Y_BOTTOM_OFFSET=  30;
  ...

  /** Run the program as an application. Parameter
      args is not used.  */
  public static void main(String[] args) {
    String[] sizeArgs= { "width=" + WIDTH,
                         "height=" + HEIGHT };
    new Breakout().start(sizeArgs);
  }

  /* Run the Breakout program. */
  public void run() {
    // Initialize and play the game.
    ....
  }
}
```

The acm package

This assignment uses the same package used in A5, so we don't say much about it. Here are API specs for the package: http://jtf.acm.org/javadoc/student/index.html. So that you can study these packages and classes in them while you are not on the internet, we provide the html files in downloadable form on the course website, but not all the images are included. Click on docs/index.html to see the documentation.

Class `Breakout` is a subclass of class `GraphicsProgram`. Method `main` creates an instance of the class and then calls method `start` of the instance, which is inherited from `GraphicsProgram`.

`GraphicsProgram` also declares procedure `run`, which is overridden in class `Breakout`. Execution of `start(sizeArgs)` constructs the basic GUI window and then calls procedure `run`. Your job is to write the body of procedure `run` (and any other methods you need) to initialize the GUI with bricks and paddle and ball and then play the game.

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 sections describe a reasonable, staged, approach to the problem. If you follow our advice and test each piece thoroughly before proceeding to the next, you should be successful.

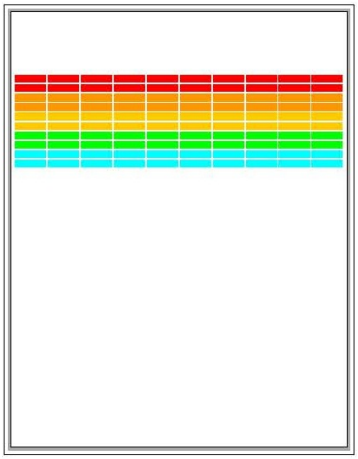
An important part of your programming will be to develop new methods whenever you need them, in order to keep each method small and manageable. You *must* precisely specify each method you write, and you should do this before writing the method body. We will not accept assignments do not provide a class invariant or do not specify methods precisely and thoroughly. You need not write loop invariants, although doing so will help you.

What to download

File breakout.zip, on the course webpage for this assignment, contains everything you need, including file Breakout.java. Download it, unzip it, and load file Breakout.java into DrJava. In folder docs, click on index.html to load the API specifications of package acm.graphics (and a few others) into your browser.

Set up the bricks

The first step is to write the code that puts the various pieces on the playing board. Thus, it probably makes sense to implement procedure `run` as two method calls: one that sets up the game and one that plays it. An important part of the setup consists of creating rows of bricks at the top of the game, which are shown on the next page.



The number, dimensions, and spacing of the bricks, as well as the distance from the top of the window to the first line of bricks, are specified using named constants given in class `Breakout`. 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. Hint: the leftmost brick should be placed at x-coordinate `BRICK_SEP_H/2`. The colors of the bricks remain constant for two rows and run in the following sequence: `RED, ORANGE, YELLOW, GREEN, CYAN`.

All objects placed on the playing board are instances of subclasses of abstract class `GObject`. The bricks and paddle are objects of subclass `GRect`. Look at class `GRect` and find a constructor that allows you to create a `GRect` object at a particular position, width, and height. Use it.

To add `GRect` object `r` (say) to the playing board, call inherited procedure `add(r)`. To start off, you might want to create a single `GRect` object of some position and size and add it to the playing board, just to see what happens. Then think about how you can place the 8 rows of bricks.

You need to fill a `GRect` with a particular color and also to set its color so that its outline is the same color (instead of black). Look for methods to do this. In order to allow the object to be filled with a color, use a procedure call like `rect.setFilled(true);`.

Here is an *important* point. You do *not* need an array or Vector to keep track of all the bricks. Just add the bricks to the playing board as you create them, and do not be concerned about keeping track of them. Later, you will see why you don't need to keep them in an array or Vector.

Here is another important point. Make sure your creation of the rows of bricks works with any number of bricks in each row —1, 2, ..., 10, and perhaps more. Remember, static variable `NBRICKS_PER_ROW` contains the number of bricks per row, and `NBRICK_ROWS` contains the number of rows —it may help when testing to make this less than 10.

Here's a suggestion: Get this part of the program working by November 24 or so, so that you can produce just the diagram shown above to the right. This will give you considerable confidence that you can get the rest done. And you'll be well on your way before time gets short.

Create the paddle

Now create the black paddle. You will need to reference the paddle often, so declare it as a private field of type `GRect`. A public static variable of class `Breakout` gives its vertical position relative to the bottom of the window.

Here's how to make the paddle track the mouse (horizontally only). Declare a procedure like this:

```
/** Move the horizontal middle of the paddle to the x-coordinate of the
    mouse position --but keep the paddle completely on the board. */
public void mouseMoved(MouseEvent e) {
    GPoint p= new GPoint(e.getPoint());
    ...
}
```

So, you are overriding inherited procedure `mouseMoved`. This procedure is called whenever the mouse is moved. Parameter `e` has a function `getPoint`, and the assignment we give you in the body stores in `p` the coordinates of the point where the mouse currently is. Replace the three dots by code that changes the x-coordinate of the paddle as indicated in the specification, using the x-coordinate of point `p`. Be careful that the paddle stays completely on the board even if the mouse moves off the board. Our code for this is 3 lines long; it uses function `Math.min` and `Math.max`.

Why don't you get this part of the program working by Saturday, 28 November?

Create a ball and make it bounce off the walls

You are now past the "setup" phase and into the "play" phase of the game. A ball is 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 probably want a private field `ball` of type `GOval` to contain the ball, since you will have to refer to it often. Keep in mind that the coordinates of the `GOval` specify the upper left corner and not the center of the ball.

The play phase of the game should contain a loop, each iteration of which (1) moves the ball a bit, (2) changes direction if it hits a wall, and (3) pauses for 10 milliseconds, using `pause(10);`. The program needs to keep track of the velocity of the ball, which consists of its horizontal (x) component and its vertical (y) component, which you declare as fields like this:

```
private double vx, vy; // give their meaning in a comment!
```

These velocity components represent the change in position that occurs on each time step (each loop iteration).

Initially, the ball should head downward, so use a starting velocity of +3.0 for `vy`. The game would be boring if every ball took the same course, so choose component `vx` randomly. You can read about random numbers in the text, but for now, simply do the following:

1. Declare a field `rgen`, which will serve as a random-number generator:

   ```
   private RandomGenerator rgen= new RandomGenerator();
   ```

2. Initialize variable `vx` 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.

Your next challenge is to get the ball to bounce off the walls of the playing board, ignoring entirely the paddle and the bricks. To do this, after moving the ball one step, by `(vx,vy)`, do the following. Suppose the ball is going up. Then, if any part of the ball has a y-coordinate <= 0, the ball has reached the top and its direction has to be changed so that it goes down, by setting `vy` to `-vy`. Check the other three sides in the same fashion. When you have finished this, the ball will bounce around the playing board forever —until you stop it.

You have to figure out whether the ball has reached (or gone over) the other three sides. Remember that the location of a `GOval` is the top-left corner of its bounding box.

Checking for collisions

Now comes the interesting part. In order to make `Breakout` into a real game, you have to be able to tell when the ball collides with another object in the window. As scientists often do, we make a simplifying assumption and then relax the assumption later. Suppose the ball were a single point `(x,y)` rather than a circle. The call

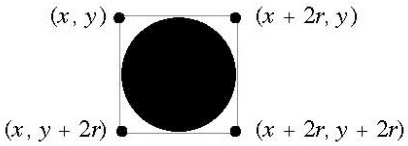
```
getElementAt(x,y)
```

of inherited function `getElementAt` does this: (1) return **`null`** if no graphical object covers point `(x,y)` and (2) return the (name on the tab of) the `GObject` that covers point `(x,y)`, if some `GObject` actually covers the point. (If several `GObject`s cover the point, the one that appears to be in front on the display is returned.)

Suppose `getElementAt` returns (the name on the tab of) a `GObject gob`. If `gob == paddle`, you know the paddle has collided with the single-point ball at `(x,y)`. If `gob != paddle` and `gob != null`, then `gob` must be a brick, since the only objects on the board should be bricks and the paddle. So, we have just explained how to test whether a single-point ball has collided with the paddle or a brick.

But the ball is not a single point. It occupies physical area, so it may collide with something on the screen even though its center does not. The easiest thing to do —which is 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.

The easiest thing to do is to check the four corner points on the square in which the ball is inscribed. 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 are at the locations shown in the diagram to the right (r is the radius of the ball). These points have the advantage of being outside the ball, so that `getElementAt` can't return the ball itself. But they are close enough to make it appear that a collision has occurred.



So, you need to check the four corners, one at a time; if one of them collides with the paddle or a brick, stop the checking immediately and proceed as if a collision occurred. We suggest that you write a function

```
GObject getCollidingObject()
```

that returns the object involved in the collision with the ball (or **`null`** if none). Call this function once, each time the ball is moved.

If the ball going up collides with the paddle, don't do anything. If the ball going down collides with the paddle, then negate the vertical direction of the ball. If the ball (going in either vertical direction) collides with a brick, remove the brick from the board and negate the vertical direction. Inherited function `remove(gob)` can be used to remove object `gob`.

Finishing up

The basic parts are now done. A few more details have to be taken care of, and these may cause you to reorganize procedure `run`. Think carefully about the following tasks and how you want to design the final program before proceeding.

1. *Take care of the case that the ball hits the bottom wall*. Right now, the ball just bounces off this wall like all the others, but hitting the bottom wall is supposed to mean that the ball is gone. In one game, the player should get three balls before losing. If the player can have another ball, put a message on the board somewhere (use a `GLabel` object), telling the player that another ball is coming in 3 seconds, pause for 3 seconds, remove the message, and continue with a new ball.
2. *Check for hitting the last brick, in which case the player wins*. An easy way to do this is to keep track of how many bricks are left on the board; when there are none, the game ends and the player has won.
3. *When a game ends*, place a message somewhere on the window (use a `GLabel` object).
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?
5. 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, try this: 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.

Strategy and tactics

Here are some survival hints for this assignment:

1. *Start as soon as possible.* If you wait until the day before this assignment is due, you will have a very hard time completing it. If you do one part of it every 2–4 days, you will enjoy it and get it done on time. The hard part may be "finishing up" —designing the final reorganization in order to incorporate three balls in a game.
2. *Implement the program in stages*, as described in this handout. Don't try to get everything working all at once. Make sure that each stage is working before moving on to the next stage.
3. *Set up a milestone schedule.* We have suggested some milestones for you, but make up your own schedule, and leave time for learning things and asking questions. There may be points about package acm.graphics that you will have to learn by yourself.
4. *Don't try to extend the program until you get the basic functionality working* (see the next section). If you add extensions too early, debugging may get very difficult.

Possible extensions

There are many ways —and fun ways— to extend this assignment. Add more and we will be a bit more lenient with grading than if you implement the bare minimum. But note that a submission that does not have a good class invariant and good specifications for the methods will not be looked at kindly under any circumstances.

1. *Let the player play as many games as they want*. The player could click the mouse button to start a new game. A call on inherited procedure `waitForClick()` will wait (or pause) until the mouse is clicked.

2. *Play a sound whenever the ball hits a brick.* This is an easy extension. Breakout.zip contains an audio file bounce.au. You can load it by writing

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

   and play it by calling `bounceClip.play();`. The sound might get obnoxious after a while, so figure out a way to let the user turn it off (and on).

3. *Improve user control over bounces.* The program gets rather boring if the only thing the player has to do is hit the ball. Let the player control the ball by hitting it with different parts of the paddle. For example, suppose the ball is coming down toward the right (or left). If it hits the left (or right) 1/4 the paddle, the ball goes back the way it came (both `vx` and `vy` are negated)

4. *Add in the kicker.* The arcade version of `Breakout` lures you in by starting off slowly. But as soon as you think you are getting the hang of things, the ball speeds up, making life more exciting. Implement this in some fashion, perhaps by doubling the horizontal velocity of the ball on the seventh time it hits the paddle.

5. *Keep score.* Display the score (number of bricks destroyed) underneath the paddle, but remember, a `GLabel` is an object, and the ball hitting it should have no effect. Perhaps you can make the bricks in the higher rows more valuable.

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

Submitting your assignment

Before the due date, (1) Put a comment at the top of class `Breakout` that contains the time you spent on this project, and also give a brief overview of any extensions you added to it. (2) ADD a comment that gives your opinion of this assignment --whether it was educational, interesting, exciting, loads of fun, to easy, too long, horribly described, how we could improve it, etc.

Submit file Breakout.java on the CMS.