BREAKOUT™

That's right, you are going to code the classic arcade game *Breakout*. The picture to the right is a screen shot of the original arcade game so you can get a feel for how it looked back in the day when it cost you a quarter to play. The goal of the project is two-fold:

- to bring all the concepts we've learned this year together to create a real-world application with animation and user control
- **TO HAVE FUN!**

You are ready for this! We have covered all the concepts you will need to use in previous labs. I have broken the project up into several smaller units to make it more manageable; you will be doing them over the next several classes. If you work on the project unit each day in class and make sure each unit is working before proceeding to the next, you will see the game coming together before your eyes! You will end up with a real game you can package into a standalone program anyone anywhere can play.

Before You Get Started

Necessary Files

There are only 3 starter files for this project and they are in our directory on the server:

`Breakout_Shell.java`
: This is the source code that you will modify, and the only file that you submit for a grade.

`acm.jar`
: This Java ARchive contains all the acm classes.

`bounce.wav`
: A sample sound file should you wish to add sound effects as part of your extensions.

You will want to refer often to the complete API for the ACM graphics package available at:

http://cs.stanford.edu/people/eroberts/jtf/javadoc/student/index.html

The following are the areas of the ACM API you will need to refer to:

- `acm.graphics`: all graphics object related methods
- `acm.program`:
  - `GraphicsProgram`: methods you ***will*** need to use
  - `Program`: methods you ***might*** need to use

Another resource you might find helpful is the ACM tutorial put together by the Java Task Force which you can find at:

http://cs.stanford.edu/people/eroberts/jtf/tutorial/index.html

---

All the source code is contained in `Breakout_Shell.java`. Your first task is to replace the 2 references to "`Shell`" in `Breakout_Shell` with your initials. Then save the file with the new name so your source file name matches the internal class name as required by Java. A partial skeleton of this class appears to the right. The class includes:

```
import acm.graphics.*;
...

public class Breakout_Shell extends GraphicsProgram
{
   // main method -- called when the program is run
   public static void main(String[] args)
   {
      String[] sizeArgs = { "width=" + WIDTH, "height=" + HEIGHT };
      new Breakout_Shell().start(sizeArgs);
   }
   // init method -- automatically called on startup
   public void init()
   {
      createBricks();      // create the bricks
      createPaddle();      // create the paddle
      createBall();        // create the ball
                           // add a mouse listener
   }
   // run method -- automatically called after init
   public void run()
   {
      startTheBall();
      playBall();
   }
}
```

Import statements for the various packages
Recall from class that a GUI application needs to access a lot of packages to function properly; this class imports them all.

Constants that control the game parameters
These are mnemonic constants for things like the height of the game paddle. Your code should use these constants internally so that changing them in one location in your file changes the behavior of your program accordingly.

A static `main` method
This is a method that starts the application program. This application treats the entire program as one object as opposed to our previous programs that would have used different classes for each object such as bricks, ball, and paddle. For this project we will be using the ACM Graphics package. The Association for Computing Machinery created this library to help make graphics programming more accessible to beginners. The ACM library incorporates many different classes which creates a one-stop shop for all the methods you would need to create a basic graphics program. One way to view the ACM approach is to think of the window (canvas) as a felt board. Objects are created and then stuck (added) to the felt board. The figure to the right shows the hierarchy for the `GraphicsProgram` that Breakout extends and all the classes it includes. This gives you access to all that functionality!

```
java.lang.Object
  └ java.awt.Component
      └ java.awt.Container
          └ java.awt.Panel
              └ java.applet.Applet
                  └ javax.swing.JApplet
                      └ acm.program.Program
                          └ acm.program.GraphicsProgram
```

Therefore, this program will work a little differently than others we have done. Class `Breakout` is a subclass of class `GraphicsProgram` which is associated with the window on which the graphics takes place. Method `main` creates an instance of the class and then calls method `start` of the instance, which is inherited from `GraphicsProgram`. You will not need to make any changes to `main` given the magic of the `init` and `run` methods. Open the acm API and click on `acm.program` to see the methods summary for `GraphicsProgram`. Note the entries for these two special methods:

`void`	**`init`**`()` Specifies the code to be executed at startup time before the `run` method is called.

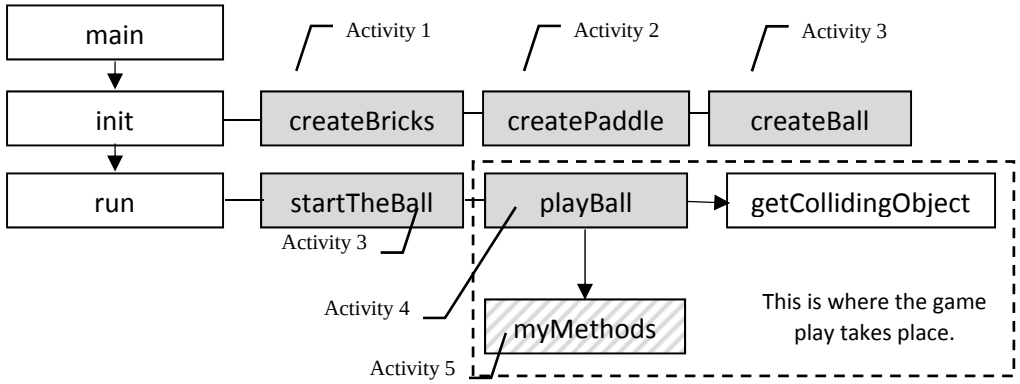
`void`	**`run()`** Specifies the code to be executed as the program runs.

In the `main` method, the instantiation of the Breakout object by the `Breakout.start(sizeArgs)` method constructs the basic GUI window and then calls the `init` method. In `init` you will put the initialization items that you want created before the program starts running (e.g. initialization of variables, labels and graphics). Next `run` is automatically called. In `run` you will call the sequence of methods you need to keep the game going. Initially, this may just be a method called `play`, but as you add features you may add other methods to `run`. As the project progresses, you will be writing the methods to initialize the GUI with bricks, paddle, and ball and manage the game play. You also have to write class `Brick` (within the Breakout class) that will be the basis for all your brick objects. An important part of your programming will be to develop new methods whenever you need them.

This figure diagrams the basic flow of the program.

`main` instantiates Breakout which calls `init` (which sets up the game) and then `run` (which starts the ball and keep game play going).

Once you have completed the methods shaded gray, you will enhance the program by modifying those methods and/or adding your own.



Program flowchart

How to Succeed with this Project

Remember to focus and work on your project in class. Make sure that each stage covered by an activity is working properly before moving on to the next activity. *Do not try to get everything working all at once*. Leave time for learning things and asking questions. Since this is the first time we are using the ACM library, you will have to learn some things about the `acm.graphics` package by yourself using the API. Finally, do not try to extend the program until you get all of the basic functionality working. If you add extensions too early, debugging may get very difficult.

John Champe Code of Honor

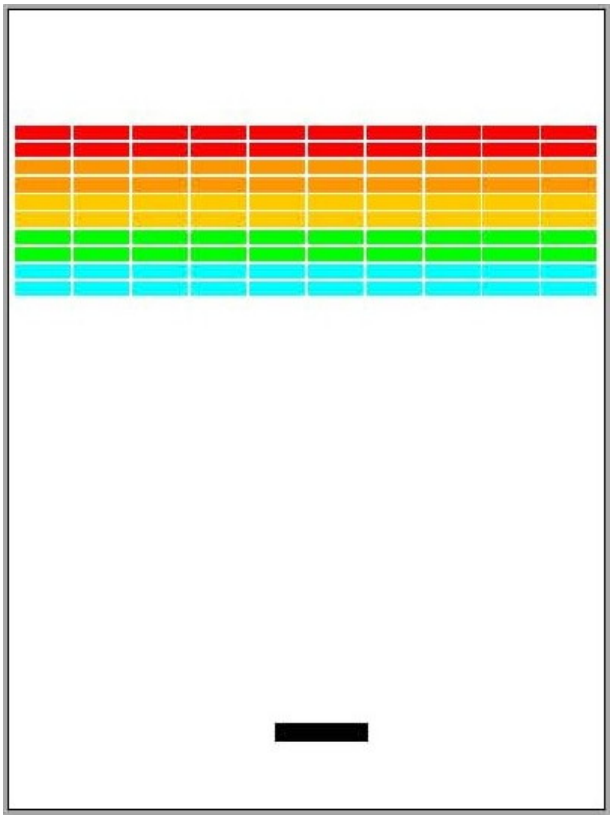
The JCHS Honor Code states that: “cheating includes, but is not limited to the following:

- giving or receiving of any work other than your own
- copying another person’s work or allowing another to copy your work”

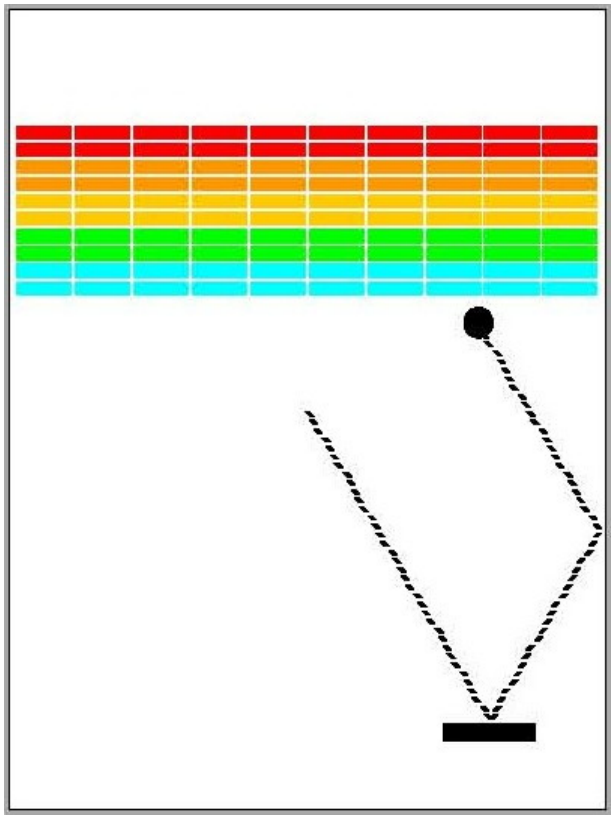
Therefore, it is a violation of the JCHS Code of Honor for you to be in possession of or to look at code (in any format) for this assignment from anyone else. You may not show or share your code electronically with another student. As with any programming assignment, it is highly unlikely that your code will look exactly like someone else's. Mr. R will be checking for instances of plagiarism which includes code taken from sources outside of JCHS. Violations of the Honor Code will receive a zero for the project. Mr. R will try to give you all the help you need to succeed at this project.

Welcome to Breakout

The initial configuration of the game Breakout is shown in the left-most picture below. 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 vertical position — it moves side to side across the screen along with the mouse — stopping at the edge if the mouse goes past the edge of the window.



Starting Position

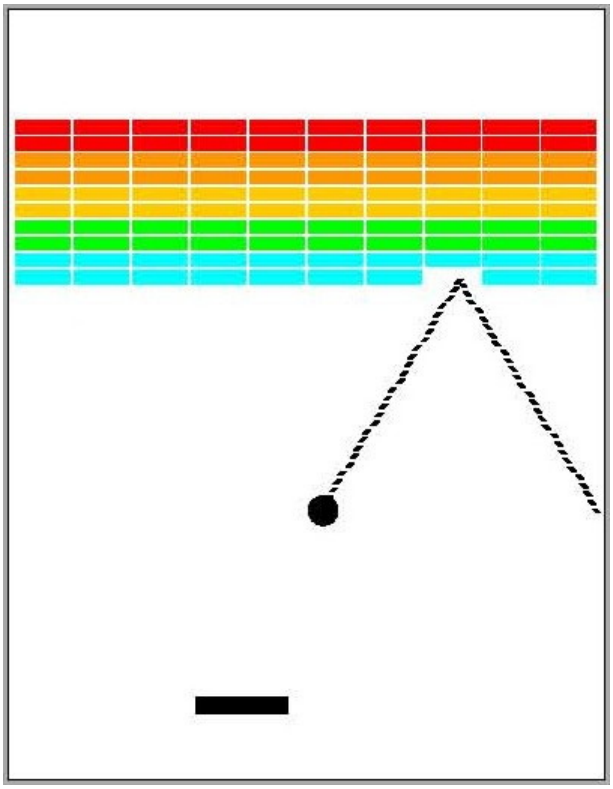


Bounced Ball About to Hit a Brick

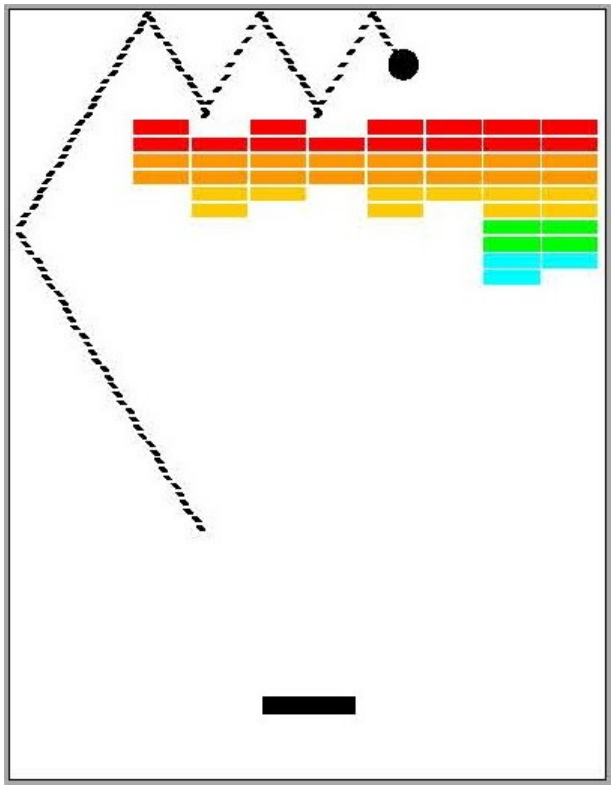
A complete game consists of three turns. On each turn, a ball is launched from the center of the window toward the bottom of the screen at a random angle. The ball bounces off the paddle and the walls of the world in accordance with the physical principle of simple reflection:

The angle of incidence equals the angle of reflection

The start of a possible trajectory, bouncing off the paddle and then off the right wall, is shown in the figure on the right. (Note: the dotted line is there only to show the ball's path and will not actually appear on the screen.) 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 left figure below shows the game after that collision and after the player has moved the paddle to put it in line with the oncoming ball.



Intercepting the Ball



Breaking Out

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

1. *The ball gets past 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.*
   In this case 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 may 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 last diagram above 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 game, but you do not have to do anything in your program to make it happen. The game is operating by the same rules it always applies: bouncing off walls, clearing bricks, and obeying the "laws of physics".

The Basic Game

I have divided the project into smaller more manageable pieces. And in turn, these pieces are arranged into 2 groups: the first part covers the basic things that you must implement just to get the game running; the second part entails possible enhancements you may choose to add to the game.

Setting up the Bricks

The first step is to write the code that puts the various pieces on the playing board (canvas). As we discussed previously, the `init` and `run` methods are automatically called for you. The `init` method handles all the setup for you; here you put all the initialization methods to create the bricks, ball, and paddle as well as create any listeners and even instantiate labels. In the `run` method you put all the method calls which drive the game.

An important part of the setup consists of creating rows of bricks at the top of the game shown in the figure. Constants in the `Breakout` class specify: the number, dimensions, and spacing of the bricks; the distance from the top of the window to the first line of bricks. The colors of the bricks remain constant for two rows and run in the following sequence: `RED`, `ORANGE`, `YELLOW`, `GREEN`, `CYAN`. If there are more than 10 rows, your start over with RED, and do the sequence again.

All objects placed on the playing board are instances of subclasses of abstract class `GObject`. The bricks and paddle are objects of subclass `GRect`. The paddle is created directly from the `GRect` class. We could also just create bricks using the `GRect` class, but instead we will base them on a `Brick` class which extends `GRect`. Why is this approach preferable?

Class `Brick` initially has no fields and needs only two methods (its constructors):

- 2-arg constructor for a new brick of a given width and height
- 4-arg constructor for a new brick at a point (*x*,*y*) and of a given width and height.

Class `GRect` has similar constructors, if you wish to compare. To add a `GRect` object, `r`, to the playing board, call `add(r)`, a method inherited by the class `Breakout` from the ACM class it extends, `GraphicsProgram`. Therefore, you will need to put a call to `add( )` in the `createBricks` method.

By default, the shell program will create solid black bricks. You need to fill a `Brick` (which is a `GRect`) with a particular color according to the color scheme mentioned above. You also need to set its color field so that its outline is the same color (instead of black). Look for methods in the `GRect` class to do this. In order to allow the object to be filled with a color, you first need to call `setFilled(true)`.

Important Considerations

Do not use an array to keep track of the bricks. Just add the bricks to the playing board as they are created. Don't worry about keeping track of them; you will see why you do not need to keep track of them later. Also, make sure your creation of the rows of bricks works with any number of rows and any number bricks in each row.

Activity 1: Touring & Brick Laying

Tour the Shell Code

Take some time to familiarize yourself with the Shell code. The better you understand the layout of the code now, the easier it will be for you to figure out where changes need to be made to accomplish what you need to do (requirements) and want to do (enhancements).

If you haven't yet, copy the files locally to your flash drive and replace "Shell" in the Breakout_Shell.java file with your initials. Remember to replace it in the class name and in the call in the `main` method as well.

Now tour the code and answer the following questions:

1. Between brick, paddle, and ball, which object has the most constants defined? Why?
2. How many methods are in the application (including main)?
3. How many classes are defined in the application?
4. What listeners are you going to need? What other listeners might you want to activate? Why?

Tour the ACM API

Visit and bookmark the links given above for the ACM API and the JTF ACM Tutorial. Play with the API and make sure you can find the methods for `acm.graphics` and `GraphicsProgram`.

Complete Method `createBricks`

While you are working with the program, you may want to play with just 1-2 rows of bricks with just 3-4 bricks per row as this will speed up testing. You see more quickly whether the program works correctly when the ball breaks out (gets to the top of the canvas) and the actions when someone wins or loses. To do this, simply change the values of the static constants for the number of rows and number of bricks in a row.

All you need to do is to produce the brick diagram shown above. This will give you considerable confidence that you can get the rest done. To complete this method, you need to:

1. Specify the limits for the nested for loops so that it sets up the rows and columns of bricks.
2. Complete the `Brick` constructor to create the bricks. The process is outline in the activity that follows titled: *Another Brick in the Wall*
3. Once you complete the constructor, then for each new brick, you need to:
    a. Enable it to be filled with a color.
    b. Specify the correct color based on the row it is in.
    c. Add the newly colored brick to the canvas.

Once you are done, your program should compile and run. Do you see the brick pattern on the screen? If so you are done laying bricks. Congratulations, you are on your way to breaking out!

Another Brick in the Wall

`createBricks`

1. What is the color sequence for the brick rows from top to bottom? _____

   _____

   _____

   _____

   _____

   _____

   _____

   _____

   _____

   _____

   _____

2. What are the parameters for the `Brick` constructor? In other words, what are the 4 argument values the 4-arg constructor is expecting to receive in order to create a brick?

   _____, _____, _____, _____

   You need to figure out what each argument is in terms of the defined brick constants:

   `NBRICKS_PER_ROW, NBRICK_ROWS, BRICK_SEP, BRICK_WIDTH, BRICK_HEIGHT,` and `BRICK_Y_OFFSET`

   Use the diagram on the next page to help you. First fill in all the blank lines using the defined constants and label the point at the upper left corner. Then work through the following steps.

   What is the initial *x* value of point A (the location of the 1st brick in the 1st **row**)?

   A_x = _____

   Now use A_x to find the *x* value of point B (the location of the 2nd brick in the 1st **row**)?

   B_x = _____

   Now use A_x and B_x to find the *x* value of point c (the location of the 3rd brick in the 1st **row**)?

   C_x = _____

   Note what you are adding each time you go to the next brick in the **row**.

   You have to fill a grid of several rows, each with columns, so you will be calling the `Brick` constructor in a nested `for loop` structure. First we will look at the inner loop which will build the columns in each row.

Assume the loop to build the columns in each row has the form:

```
for(int c = 0; c < NBRICKS_PER_ROW; c++)
```

Note that the top range for the loop is the number of bricks per row which is the number of columns. How do you write the final value for *x* using the loop control variable, `c`?

`x =` _____

Now let's repeat the process for *y*. We know A_x will be the *x* value of every brick in the first column. What is the initial *y* value of point A (the location of the 1st brick in the 1st **column**)?

A_y = _____

Now use A_y to find the *y* value of point D (the location of the 1st brick in the 2nd **column**)?

D_y = _____

Now use A_y and B_y to find the *y* value of point E (the location of the 1st brick in the 3rd **column**)?

E_y = _____

Note what you are adding each time you go to the next brick in the row.

Now let's look at the outer loop which will build the rows. Assume the loop to build the rows has the form:

```
for(int r = 0; r < NBRICKS_ROWS; r++)
```

Note that the top range for this loop is the number of rows in the grid which is the number of columns. How will the final value for the *y* argument in the constructor be stated:

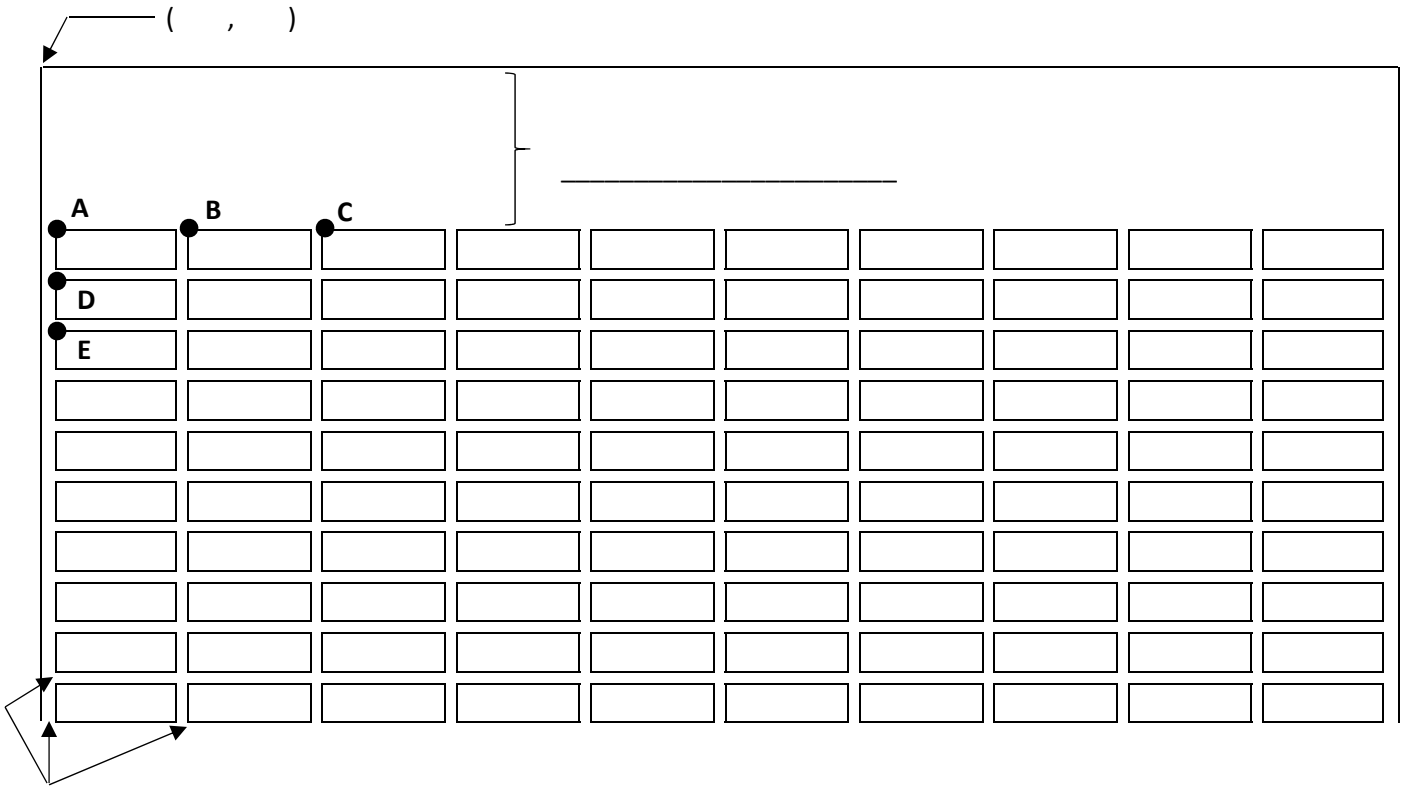
`y =` _____

Now for the easy ones! What are the width and height arguments of the `Brick` constructor for each brick, expressed in terms of the defined constants?

`w =` _____

`h =` _____

Now you have the values to plug into the `Brick` constructor which sits inside the nested for loops which will create the brick grid! Plug it in, compile and run the program; you should see the `NBRICKS_ROWS x NBRICKS_PER_ROW` grid of black bricks.


(,)
A
B
C
D
E

_____