AP Computer Science AB

Final Project: Breakout

The Shape of Things to Come

In this lab, you'll be writing the classic arcade game Breakout. Go ahead and download ShapeDisplay.java, a simple class capable of displaying a number of colored rectangles and circles. In order to compile `ShapeDisplay`, you'll first need to implement the class `Shape`, which should support the following methods.

`Shape` Class

```
public Shape(boolean isRound, Color color,
             double x, double y,
             double width, double height)
public boolean isRound()
public Color getColor()
public double getX()
public double getY()
public double getWidth()
public double getHeight()
```

You should now be able to compile the `ShapeDisplay` class.

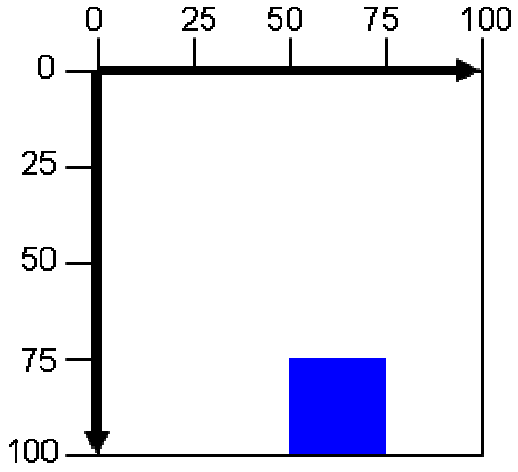
Pretty Pictures

Create a new class called `Breakout`. In the constructor, make a new `ShapeDisplay` and store it in an instance variable. Set the title of the display window to be "Breakout".

`ShapeDisplay` Class

```
public void setTitle(String title)
public void setBackgroundColor(Color color)
public void add(Shape shape)
public void repaint()
public Iterator<Shape> shapes()
public void addKeyListener(KeyListener listener)
```

The `ShapeDisplay` window considers (0, 0) to be the upper left corner, and (100, 100) to be the lower right. The following diagram shows a blue square that's been added to position (50, 75) (the position of the shape's upper left corner), with side length of 25.



In the `Breakout` constructor, try adding a couple shapes to your display window (temporarily).

Having a Ball

Create a subclass of `Shape` called `Ball`. `Ball`'s constructor should take in *just two arguments*: an x-coordinate and a y-coordinate, representing the top left corner of the ball. The ball should have a diameter length of 2, and would look great in white. In the `Breakout` constructor, create a `Ball`, stick it in an instance variable, and add it to the display at position (50, 80). (If you added other shapes, you'll probably want to get rid of them now.) You should now see a white circle on your display.

What Goes Up ...

We'd like to make our ball move on the display, and thus we'll need to be able to change its *x* and *y* coordinates. But the `Ball` class cannot directly change its coordinates, since they're hidden in the `Shape` superclass. So, go ahead and add the following methods to the `Shape` class.

```
public void setX(double newX)
public void setY(double newY)
```

A ball is a physical object, so it ought to behave roughly like the objects we studied in physics class. In particular, unless acted upon by a force (and we have no forces in our game yet), an object will maintain a constant velocity. Thus, not only should the ball move, but *it should remember how fast it is moving, so that it may continue to move at the same velocity.*

Recall from physics class that velocity refers to both an object's speed *and* its direction. So, we could store the velocity as a vector in polar coordinates, with a magnitude (speed) and angle (direction). But this would be a nuisance, since we also know from physics class that we'll spend a lot of time resolving it into its *x* and *y* components. Therefore, it will be much simpler to store the ball's velocity in rectangular coordinates, *keeping track directly of the x and y components of the velocity*. Therefore, you should go ahead and add the following two instance variables to the `Ball` class.

```
private double velX;
private double velY;
```

In the constructor, we'll initialize the `Ball`'s velocity so that the *x* component of its velocity is 0.309 and the *y* component is -0.951 (causing the ball to move up and to the right at a speed of 1 unit).

Now add the following `move` method to the `Ball` class. This method should handle the ball's movement in one instant of time, by causing the ball's *x* coordinate to change by `velX` and *y* coordinate to change by `velY`.

```
public void move()
```

Finally, add a `play` method to your `Breakout` class, which should repeatedly (forever) ask the ball to `move`, call the display's `repaint` method (so that the display knows to draw the ball in its new location), and pause for 25 milliseconds using the following code. Moving 1 unit every 25 milliseconds, it will take the ball about 2.5 seconds to cross the display.

```
try { Thread.sleep(25); }
catch(InterruptedException e) { /* ignore it */ }
```

Test your `play` method, and you should see the ball soar up and out of view.

... Can Go Through Walls?

Create a new subclass of `Shape` called `Obstacle`. So far, `Obstacle`s will be just like `Shape`s, except that they are *always* rectangular (never round). In the `Breakout` constructor, add an `Obstacle` at position (0, 0) with width 4 and height 100, so that it appears as a wall along the lefthand side of the display. Now add another obstacle with the same dimensions along the top and righthand sides of the display. (White walls would look nice.) Test the `play` method now, and watch as the ball soars right through your new obstacles!

Bouncing Off the Walls

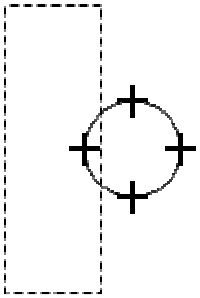
Now its time to implement the heart of our Breakout game: bouncing! Clearly, in order for the ball to start bouncing, we'll need to change its velocity. Since, in the game of Breakout, different obstacles cause the ball to change its velocity in different ways, we'll put obstacles in charge of changing the ball's velocity. Therefore, we'll need to give `Ball` the following methods, to grant obstacles access to the ball's velocity.

```
public double getVelocityX()
public double getVelocityY()
public void setVelocityX(double newVelX)
public void setVelocityY(double newVelY)
```

Next, we'll need to check if the ball has collided with an obstacle. To do so, it will first be useful to add the following helper method to the `Obstacle` class. The `containsPoint` method returns `true` if (and only if) the given point is contained within the `Obstacle`'s boundaries. (Your game will probably play best if you consider an `Obstacle` to contain the points that comprise the boundary itself.)

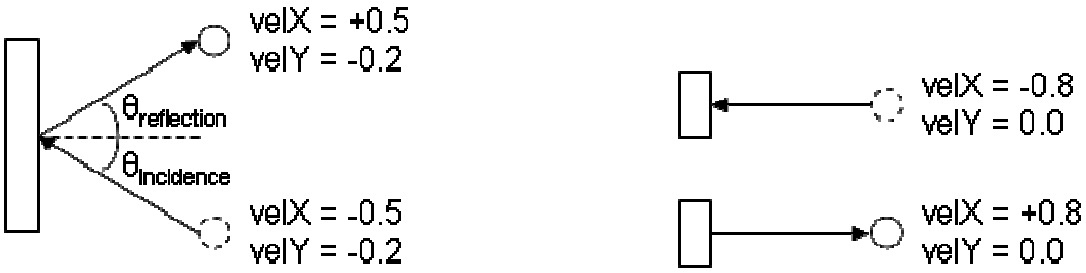
```
private boolean containsPoint(double x, double y)
```

Next, use the `containsPoint` method to write the `Obstacle` method `overlapsWith`. Here, we'll take the easy way out, and just check if the obstacle contains the topmost, leftmost, rightmost, or bottommost points of the ball (as shown in the diagram below). This may seem like a rather suspicious approximation, but our ball is small enough that this formula will work fine in practice.



```
public boolean overlapsWith(Ball ball)
```

Recall from physics class that the angle of incidence in a collision equals the angle of reflection (in a perfectly elastic collision with no spin, friction, etc). In the example collisions below, notice that the *x* component of the ball's velocity has changed signs in response to the collision, indicating that the ball is now traveling in the opposite direction. On the other hand, the *y* component of the ball's velocity is unchanged by the collision. (A collision in a different direction will result in analogous changes to the velocity.)

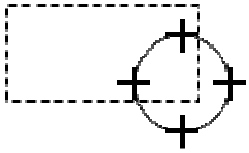


Our next task will be to write the method `handleCollision` to the `Obstacle` class. This method will only be called when a collision between the ball and the obstacle *has already been detected*. Don't write it until you've read the next several paragraphs.

The `handleCollision` method must detect which part of the ball has struck the obstacle, as this will determine which way the ball should bounce. Again, we will only consider the uppermost, leftmost, rightmost, and bottommost points on the ball. For example, if the leftmost point of the ball has struck the obstacle, then the ball was probably traveling to the left, and therefore should bounce off to the right (with the *x* component of its velocity changing from negative to positive).

Sometimes, however, you'll find that, even though the leftmost point of the ball is contained in the obstacle, the ball was actually traveling to the right. (This might happen because the ball is too puny or the obstacle was also moving, etc.) If you simply invert the sign, you'll cause the ball to move *toward* the obstacle and start jiggling around inside it. (Such a bug is sometimes called a *glitch*.) It would therefore be a good idea to make sure the ball heads off to the right, *even if it was already moving to the right*. (You may find the `Math` class's `abs` method useful here.)

Finally, don't assume that only one of the ball's four points will be contained within the obstacle. For example, it will sometimes happen that *both* the leftmost and the topmost points of the ball are embedded in an obstacle, as pictured below. Since the leftmost point has struck the obstacle, the ball should bounce off to the right. However, since the topmost point has also struck the obstacle, the ball should bounce off downward. The net effect of dealing with each of these points separately should be that the ball bounces off down and to the right.



Go ahead and implement the `handleCollision` method. It should return `false`. (We'll see why shortly.)

```
public boolean handleCollision(Ball ball)
```

In the `Breakout` class, add a helper method `checkForCollisions`, which should ask the display for each of its shapes. If the shape is an `instanceof Obstacle`, check if the obstacle overlaps with the ball, and if so, handle this collision.

Now use the `checkForCollisions` method to check for collisions whenever the ball moves. You should now see your ball bouncing off the walls and disappearing below the display.

Without a Paddle?

This game would be a lot more fun if we could use a paddle to keep the ball from falling off the screen. Create a new class called `Paddle`. We can consider a `Paddle` to be a kind of `Obstacle`, since it also must handle collisions with the ball. On the other hand, we'll see that the paddle in a game of Breakout handles collisions in a more complex way. Therefore, we'll make `Paddle` a subclass of `Obstacle`. It's constructor should take just *two arguments*, an *x* and *y* coordinate. The paddle should have a width of 20 and a height of 5. And it's a well known fact that blue paddles are particularly attractive. (We'll save overriding the `handleCollision` method for later.)

In the `Breakout` constructor, make a new paddle at position (40, 90), store it in an instance variable, and add it to the display. Test that the paddle appears correctly, and that the ball bounces off it (if you get lucky and that ball strikes the still unmovable paddle).

A Key Move

We'd like to be able to control the paddle with the arrow keys. When a key is pressed, the `ShapeDisplay` class informs its `KeyListener`s. The `KeyListener` interface (in the `java.awt.event` package) looks something like this.

```
public interface KeyListener
{
     void keyPressed(KeyEvent e)
     void keyReleased(KeyEvent e)
     void keyTyped(KeyEvent e)
}
```

To move the paddle, we only care about the case when a key is *pressed*, and not when it is released or typed. If we make the `Breakout` class implement the `KeyListener` interface, we'll need to write all three methods, even though two of them will be empty. But an easier way is to take advantage of Java's `KeyAdapter` class (also in `java.awt.event`), which looks something like the following.

```
public class KeyAdapter implements KeyListener
{
     public void keyPressed(KeyEvent e) {}
     public void keyReleased(KeyEvent e) {}
     public void keyTyped(KeyEvent e) {}
}
```

At first glance, the `KeyAdapter` class looks a bit pointless. Although it technically implements the `KeyListener` interface, each of its methods do absolutely nothing in response to a `KeyEvent`. But now we can make `Breakout` a subclass of `KeyAdapter`. This means that `Breakout` will automatically implement the `KeyListener` interface. Furthermore, we'll now be able to override the `keyPressed` method, without ever worrying about the other two `KeyListener` methods.

In the `Breakout` constructor, use the `ShapeDisplay`'s `addKeyListener` method to register the `Breakout` game as one of the `ShapeDisplay`'s `KeyListener`s. Your code should now compile, although it won't do anything new yet.

Now go ahead and add a `keyPressed` method to the `Breakout` class. Have the method do nothing but print "`Got here`". Then run your game and test that your message appears whenever you press a key.

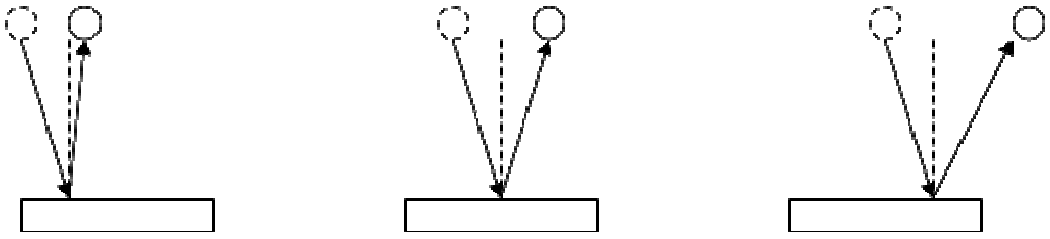
Now, instead of printing "`Got here`", check if the left and right arrow keys were pressed by calling the `KeyEvent`'s `getKeyCode` method, and testing if the `int` you get back is equal to `KeyEvent.VK_LEFT` or `KeyEvent.VK_RIGHT`. If so, move the paddle by 1 unit in the appropriate direction. Make sure to tell the display to repaint the paddle in its new location. Test that you can move the paddle with the arrow keys.

Finally, make sure your code prevents the paddle from moving through the walls. (You may assume the walls occur at positions 4 and 96.)

Breakout Physics

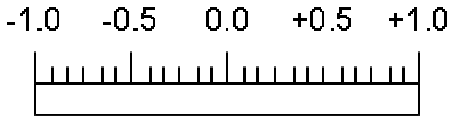
Play your game a few times, and you'll discover something funny about it. The ball seems to always bounce at the same angles! If the ball starts out moving at a steep angle (for example), it will forever bounce around at a steep angle. As a result, there is currently no way of using the paddle to *aim* the ball. This is because the `Paddle` class currently handles a collision in exactly the same way that the walls do. We'll now modify `Paddle`, so that it first deflects like an `Obstacle`, but then alters this deflection based on where the ball struck the paddle. (Don't tell your physics teacher!)

Specifically, if the ball strikes the center of the paddle, it will bounce as usual. However, if the ball strikes the lefthand side of the paddle, it should bounce a little more to the left than usual. Likewise, if the ball strikes the righthand side of the paddle, it should bounce a little more to the right than usual. The ball's contact point with the paddle should only alter its angle of reflection—not it's speed.



Begin by overriding `handleCollision`, and then asking the `Obstacle` superclass to handle the collision anyway. *Do not copy the code from `Obstacle`!* Be sure to return `false`. Test that your game still plays the same way.

Imagine that the horizonal axis of the paddle is labeled as shown. We'll call this value the *relativeX* of the collision. Thus, a collision that takes place at the center of the paddle has a *relativeX* of 0.0, a collision at the left end has a *relativeX* of -1.0, and a collision at the right end has a *relativeX* of +1.0.



After the superclass has handled the collision, have `Paddle`'s `handleCollision` method determine the *relativeX* for the collision by comparing the *x* coordinate of the *center of the ball* with the *x* coordinates of the left and right ends of the paddle. Print out the *relativeX* value, and play your game to test that you have computed it correctly.

Next we'll find the angle at which the ball is heading (following the collision). Use the `Math` class's `atan2` method (a wonderful version of arctan that does all that unit circle junk to figure out which quadrant your angle should be in), which takes in two arguments. Pass it the *y* and *x* components of the ball's velocity (in that order!), and you'll get back the ball's heading *in radians*.

We want to alter the ball's heading by no more than 15 degrees. (Once you've finished this exercise, feel free to go back and try other values.) Therefore, we will multiply *relativeX* by 15, and this will tell us how much to change the ball's angle. For example, if the ball's collision occurs at a *relativeX* of -0.5, then it's heading will be decreased by 7.5 degrees. Unfortunately, once you determine this change in angle, you will need to convert it to radians before you can find the ball's new angle.

Finally, set the *x* component of the ball's velocity to be the cosine of the adjusted angle, and the *y* component to be the sine of it. Now play your game, and practice aiming the ball. Make sure it behaves correctly. (In predicting how the ball should bounce, it may help to think of the paddle as being slightly rounded.)

(It might bother you that our coordinates are inverted relative to the ones you use in math class, and that therefore we should have minus signs scattered throughout our code. That's true, but it turns out that all those effects cancels out, so it's safe for us not to think about it.)

Bricklayer

So far we've implemented a sort of virtual one-player game of squash. In 1976, Apple Computer co-founder Steve Wozniak thought of something that would change the game of squash forever: exploding bricks! Like Wozniak, we'll add bricks to our Breakout game. Being rectangular shapes that balls bounce off of, bricks are a kind of `Obstacle`. Therefore, create a `Brick` class and make it a subclass of `Obstacle`.

The `Brick` constructor should take in a color and position, but not the dimensions of the brick. All bricks will have width 6 and height 4.

In the `Breakout` constructor, place a `Brick` at each of the following positions. (Experts agree that the top four rows look best in red, while the bottom four look great in green. Who are you to disagree with the experts?)

(5, 15)	(12, 15)	(19, 15)	...	(89, 15)
(5, 20)	(12, 20)	(19, 20)	...	(89, 20)
...	...	...	...	...
(5, 50)	(12, 50)	(19, 50)	...	(89, 50)

Now make sure your bricks appear in the right places, and that the ball bounces off them. No, they won't explode just yet.

Breaking Bricks

In the game of Breakout, when the ball strikes a brick, the brick disappears. The `Obstacle` class's `handleCollision` method returns a `boolean` value, that will tell us whether that obstacle should disappear in response to the collision. Until now, our obstacles have returned `false`, to indicate that they *should not* disappear. Now, a `Brick` should handle a collision exactly like an `Obstacle`, except that it should return `true` to indicate that it *should* disappear. Implement this behavior in the `Brick` class, *without copying code from the `Obstacle` class*!

Now modify the `Breakout` class so that it uses `handleCollision`'s return value to determine whether to remove the `Obstacle` from the `ShapeDisplay` (by way of the `Iterator`'s `remove` method).

You should now be able to play your Breakout game.

Additional Suggestions

Here are some features you might add to your game.

- multiple "lives", so that when you lose one ball and you have more lives left, a new ball appears
- a scoring system and display
- bricks that you need to hit multiple times, unbreakable bricks, bricks that break without causing the ball to bounce
- advancing to the next level when you've broken all the bricks
- bricks that alter the game when broken (perhaps by dropping items that you catch with your paddle), by: modifying the paddle size, introducing new balls, allowing you to shoot down bricks, giving you another paddle at the same time, taking away lives, granting an extra life, taking away a life, advancing to the next level, etc.
- anything you like!