COSC102 Elastic Collision Fall 2014

In this assignment, you complete a simulation of elastic collision among objects in two dimensions. The provided code is set to permit the user to populate a window with any number of balls and your code will enable those balls to move, including bouncing off of the window edges and of each other.

The goal is to learn **how to write and contribute to object-oriented programs**. The graphical user interface code that draws the window and renders the balls is provided; you are implementing the back-end for the simulation: the code that keeps track of the state of each ball.

Indeed, each ball object is an instance of **the class `Ball`, whose implementation you write**. Thanks to the `Ball class` each ball object has a position, speed and angle and instance methods that permits to update these values to keep moving it within the window and among the other balls.

Introduction

Setting up

Downloas the starter files for this assignment from Moodle. There are three files.

1. `CollisionApp.java`: the source code for the GUI (Graphical User Interface), which contains the `main` method. Run this class to execute the elastic collision application.
2. `Ball.java`: the skeleton code for the class `Ball` you implement.
3. `DemoBall.class`: a compiled Bytecode for a working version of the class `Ball`.
   Warning: This file cannot be opened by DrJava or any editor.

Download these files to a working folder, `hw3`, on your machine. Open `CollisionApp.java` and `Ball.java` in DrJava, compile them and run the application.

Running the Demo Program

`Ball.java` in addition to the skeleton code contains special code that references the implementation `DemoBall`. The purpose is for you to see how the program should work once you have finished your code.

To run the program, compile the Java source code files and run the class `CollisionApp`. It opens a window containing a "New Ball" button above a drawing area. Each time you press the button, a new ball is created using the values for the three parameters shown on the display.

1. Radius: the radius of the ball, which determines its size and mass.
2. Speed: the number of pixels per time-step that the ball moves.
3. Angle: The direction in which the ball is moving, in degrees.
   Zero degrees is moving right; 90 degrees is straight up; -90 or 270 degrees is straight down; and so on.

Edit the text boxes to change the parameters before clicking the "New Ball" button, to see their effects on the ball appearance and motion. You should notice the following features of the simulation.

- Balls reflect off the window borders. Resizing the window does change the space available for the simulation, and the balls detect collision accordingly.
- Balls bounce off of each other. The new velocities are calculated according to the rules of elastic collision. In particular, total kinetic energy is conserved.

To turn off the use of the `DemoBall` implementation and use your own code, change to `false` the `DEMO` boolean variable declaration in the file `Ball.java` (a comment describes this feature in the file). Change the value of this variable any time and re-compile to switch between the `DemoBall` implementation and your own.

Program Flow

This section describes the overall structure of the application, i.e., the relevant connections between the `BallPanel` class in the `CollisionApp.java` file and the `Ball` class defined in the `Ball.java` file.

When the program is launched, a timer object is created for the window (see the `Timer` instance in the `BallPanel` class). Each "click" of the timer triggers an update to the positions of all the balls in the simulation (see the `actionPerformed` method of `BallPanel`), after which the balls are re-drawn on the screen (see the `paintComponent` method of `BallPanel`). This is the basic process for any computer animation.

When the "New Ball" button is pressed, an instance of the class `Ball` is constructed using the parameters specified in the window. A new ball example is added to the simulation.

Each time-step the following operations are performed.

- The positions of all the balls are updated. Each ball shifts its (x, y) position according to velocity.
- If the ball is at the edge of the window, its corresponding horizontal or vertical velocity is changed to make the ball bounce back towards the inside of the window.
- Pairs of balls are checked for possible collisions.
- If there is a collision, then the balls adjust their velocities based on the rules of elastic collision.
- Finally, all the balls are drawn in the current (updated) positions.

Note that each ball keeps track of its velocity, so that it can update its own position each time-step.

Implementation

Your task is to complete the implementation of the class `Ball` to have the application working without using the `DemoBall` reference. To do so you have to write the methods declared in the class `Ball`. Do not change or remove any existing method declarations, but feel free to add helper methods.

Except for the two variables used to link to the `DemoBall` implementation, no instance variables have been declared. The first step is to think about which variables are needed to maintain a ball state and their types so as to declare them as instance variables in the `Ball` class.

Read carefully below as this description walks you through the steps to develop the implementation of the class `Ball`. **Read at least once through all the steps** to know the overall sequence needed before to actually start implementing the first step. Then in the order presented code and test each step before moving to the next one.

Instance Variables and Accessor Methods

Each instance of `Ball` has properties, some of which the `CollisionApp` program accesses through method calls (in order to draw it, to update it and so on). A ball's state is defined by:

- a radius,
- a position (x, y), the coordinate of the center of the ball
- a velocity vector (x_{vel}, y_{vel}), and
- a mass.

The velocity vector represents the rate of change in the $x-$ and $y-$directions based on the speed and angle parameters (the formulas are given below).

A ball has a mass, which is needed to implement elastic collision. For the sake of the simulation, the mass, M, is just the volume of a sphere considering the ball's radius, r, and thus is evaluated by

$$M = \frac{4}{3}\pi \cdot r^3.$$

Define instance variables to store these properties and provided the following accessor methods, as the `CollisionApp` class requires them.

- `getX()`, `getY()`: return the current $x-$, $y-$ respectively, coordinate position of the ball.
- `getRadius()`: return the radius of the ball.

Constructor

When a ball is created, the constructor is provided the following parameters by the GUI code:

- a radius,
- the speed and
- an angle.

Theses parameters are used to initialize the ball's instance variables, and in particular the velocity vector. The velocity vector represents the speed and direction (given by the angle) in which the ball travels, thus its components are computed as

$$x_{vel} = speed \cdot \cos(angle)$$
$$y_{vel} = speed \cdot \sin(angle)$$

The Math library in Java contains the trigonometric functions needed. Be aware that these functions expect angles in radians. Since the angle provided to the constructor is in degrees a conversion is necessary.

A positive $x-$velocity corresponds to moving towards the right; a negative $x-$velocity corresponds to moving towards the left.

In term of the initial position (x, y) you can use $(0, 0)$, which is the center of the window at all times, or other coordinate values as long as the ball can be seen immediately.

Note: Once you declared the instance variables and implemented the constructor and accessor methods, **compile and test your code**.

- Set the variable `DEMO` equals to `false`.
- Run the `CollisionApp` program.
- Push the `New Ball` button in the GUI (graphical user interface).

The result should be a ball in the center of the window or wherever you decided. (It won't move yet.)

Animation

Updating the Position

Now the goal is to implement the moving of the balls, without worrying about collision.

To do so the `updatePosition` method, which is called at each time-step (by the `CollisionApp` code), is completed. `updatePosition()` adjusts the position once. **Do not write a loop here**: the animation loop is managed by code outside the `Ball` class.

In `updatePosition()` write assignments to add to the current position (x, y) the $x-$ and $y-$components, respectively, of the velocity vector.

Compile and test the program. Now, the balls are moving and keep moving, even beyond the window...

Bouncing off the Window's Edges

To limit the balls within the window a change of direction is applied once a ball hits a window's edge: within `updatePosition()` the sign of a velocity component occurs each time the ball touches an edge of the window.

To do so we need to understand the coordinate system used in the window, which is defined in the class `BallPanel` contained in the `CollisionApp.java` file. The coordinate system is maintained at the center of the panel component: the center of the window is always $(0, 0)$. Since window resizing is allowed the `updatePosition` method of the `Ball` class is passed two parameters: `xmax` and `ymax`, which correspond to the most positive $x-$ and $y-$coordinates of the window current size. Thus, the legal coordinates for the ball are within `[-xmax, +xmax]` and `[-ymax, +ymax]`. To make the bounce realistic the radius of the ball has to be taken into account.

For example when the ball is at the right edge of the window the ball $x-$component of the velocity vector is adjusted when the following condition occurs.

```
if (x > (xmax - radius))

    xvel = -Math.abs(xvel);
```

The current $x-$position of the ball, `x`, is compared to the value `xmax-radius` to decide if the ball is at the farthest-right position of the window and therefore needs a sign change for its velocity vector.

Understand this example so as to determine the analogous cases for the other three edges of the window. Drawing a picture helps!

Once bouncing from the edge is implemented, compile and test your program. The balls should all stay inside the window, even after resizing the window (wait a little bit for the balls to come back into view if you shrink the size of the window.)

Collision

Finally, the problem of balls passing through each other is addressed in two stages. First, collision is detected: a boolean function reports whether or not a ball will collide with another. Then the collision response can be produced: elastic collision adjusts the velocities of colliding balls.

Detecting Collision

The instance method `isCollidingWith` needs to be completed. The method, which is called on one `Ball` given another `Ball` as a parameter, checks if these two balls are near enough to each other to collide.

To determine collision the distance between the balls is computed and compared to the sum of the two radii. To avoid taking a square root, the square of the sum of the radii can be used. Here are the formulas in terms of the (x, y) positions and radii of the two balls:

- Distance (squared): $D = (x_1 - x_2)^2 + (y_1 - y_2)^2$
- Sum of radii (squared): $S = (r_1 + r_2)^2$

If $(D <= S)$, the two balls are close enough to collide.

Unfortunately, this is not enough for the animation. Collision detection happens after all the positions are updated, and the positions do not update again until the next time-step. So, by not being careful, a collision may be detected again and undo any changes just made.

To address this problem we check if the balls are actually moving towards each other: if they are they will collide; if not, they won't. This way, once a collision is detected and the velocities changed (as described below) the balls move away from each other and the velocities are not changed again.

So, first check the condition above, and *if* the balls are close enough, then compute the following relation

$$P = (x_1 - x_2) \cdot (x_{vel_2} - x_{vel_1}) + (y_1 - y_2) \cdot (y_{vel_2} - y_{vel_1})$$

using the instance variables of the two `Ball` objects involved.

If $P > 0$, then the balls are moving towards each other.

So, **in summary the method `isCollidingWith`**:

1. computes the distances, D and S
2. if $D <= S$, then computes P
3. if $D <= S$ and $P > 0$, returns `true`
4. in all other cases, returns `false`

Elastic Collision

Finally, the best part, elastic collision is implemented: the velocities are changed when two balls collide.

In the `BallPanel` class, the `actionPerformed` method checks and update pairs of objects using the instance methods from the `Ball` class `isCollidingWith` and `collide`.
If `isCollidingWith` returns true, `collide` is called, passing also another ball as a parameter, so as to change the velocities of the colliding balls.

Below are the formulas to do this, using the same notation than above, with in addition m being the mass of a ball. First, the quantities C_x and C_y, which represent the horizontal and vertical components of the speed difference used, are computed so as to adjust the velocities after the collision.

$$C_x = (x_1 - x_2) \cdot P/D$$
$$C_y = (y_1 - y_2) \cdot P/D$$

Then, the velocities are adjusted based on these values and the mass of the two balls, such that for one ball the velocity components become

$$x_{vel_1} = x_{vel_1} + 2 \cdot C_x \cdot m_2/(m_1 + m_2) \quad \text{and}$$
$$y_{vel_1} = y_{vel_1} + 2 \cdot C_y \cdot m_2/(m_1 + m_2).$$

Similar velocity update are used for the second ball but with a negative sign so that kinetic energy is conserved:

$$x_{vel_2} = x_{vel_2} - 2 \cdot C_x \cdot m_2/(m_1 + m_2) \quad \text{and}$$
$$y_{vel_2} = y_{vel_2} - 2 \cdot C_y \cdot m_2/(m_1 + m_2).$$

Thereafter, the velocities reflect the speed and direction of the two colliding balls.

Run and test your implementation.

Final Change

On your own, without being given guidance, add code so that the color of the ball changes at the moment a collision occurs. When two balls collide they should change color, you can choose if you want to use a constant color, a random color or swapping their color.

If you want to add a color change when the ball hits a window edge, feel free to do so.

Run and test your final implementation.

Submission

Include in `Ball.java` header

- your student information,
- the number of hours you spend on this assignment,
- a short description explaining how you implemented color change,
- if you have not fully implemented the assignment a description of each part missing and
- acknowledgement crediting people and resource(s) that helped you.

Submit early and often. Late assignments are not accepted, so make sure you have some version of the source code uploaded to Moodle. You can replace the uploaded file with a more up-to-date version until the deadline.

Submit *only* `Ball.java`, the file containing the source code for your implementation of the class `Ball`, using the upload link on Moodle. Remember, it is the text file that contains your Java code and has a `.java` extension (although this extension may not be visible on all computers). Just make sure you do not submit your `.class` (compiled bytecode) or `.java~` (backup) file.