Indiana University
CSCI-C212: Introduction to Software Systems (S25)
Joshua Crotts, Muazzam Siddiqui

Even Even More Object-Oriented Programming

What To Do:

Follow each step carefully. There is no autograder for this assignment, and you'll soon understand why.

Questions

In this lab you will be working on finishing a breakout/brick breaker/Arkanoid clone. If you have never heard of this game before, please go to `https://www.coolmathgames.com/0-brick-breaker` and play it for a few minutes to get an idea of how the game works.

You will not be working with the graphics or the collision detection, as that would be too much. Your job is to design a few classes that handle the object hierarchy in the system.

Download the starter code from Canvas and import it into IntelliJ. Nothing will compile for the time being, because several classes (that you will write) are missing.

`GameObject` Class

First, design the abstract `GameObject` class, whose constructor receives four `double` values: `x`, `y`, `width`, and `height`. Store these as instance variables and write the relevant accessor and mutator methods. This class represents an *entity* in the game. Also, make `GameObject` implement the `ICollidable` interface. This interface describes objects that are collidable, i.e., can collide with one another. Hopefully, you can begin to see why we might want objects to collide with one another. An object that responds to collisions with others will override the `handleCollisions` method. In a future step, we'll see how this is utilized. `GameObject` should also declare the following abstract methods:

- `abstract void update();`
- `abstract void render(Graphics2D g2d);`

`Paddle` Class

Next, go into the `Paddle` class, and make it extend `GameObject`. This will require you to design a constructor that then calls the superclass constructor. But, the `Paddle` constructor should only receive two values: the starting *x* and *y* coordinates. For its dimensions, initialize two private, static, and final integers to denote the width and height of the paddle to be 100 by 20 pixels. Also make `Paddle` store two new fields: `velX` and `velY` representing the *x* velocity and *y* velocity respectively. When you press the left and right arrow keys, the paddle will move to the left and to the right (this functionality is already implemented into the system). Finally, make `Paddle` implement the `ICollidable` interface. This will require you to override three methods: `isPaddle`, `isBall`, and `isBrick`. Use common sense to update these to return the correct boolean value.

`Ball` Class

After this, move to the `Ball` class. You will see that it is, effectively, already complete. Change the constructor to initialize the velocity variables to be random `double` values between -10 and 10 (inclusive or exclusive - doesn't matter) rather than 2 and 5. Then, take a look at the `update` method. While there is nothing here that we want you to change or add to, take a few minutes to discuss with your partner(s) on what this code does and how it works. Then, move to the `handleCollisions` method. You will see that there is a `for` loop over a given list of `GameObject` instances. We then check to see if the current object is a paddle or is a brick. These conditionals correspond to the questions, "Does the ball collide with the paddle," and "Does the ball collide with a brick." In the former case, we perform some simple vector math to update the ball's velocity. In the latter case, we remove the brick from the list of entities and reverse the y velocity of the ball. Again, take some time to look through this code and discuss with your partners.

`Brick` Class

Lastly, finish designing the `Brick` class to also extend `GameObject` and implement `ICollidable`. This should be extremely straightforward. Same as `Paddle`, its constructor should receive only positional values, but you need to pass to the superclass constructor the dimensions. Define these as private, static, and final constants; they should be 50 and 20 respectively.

Final Thoughts

After this, everything should now compile. Go to `BreakoutRunner` and run the program. Move the paddle around with the left and right arrow keys.

This is a very small lab, but hopefully you now understand the purpose of abstract classes: they serve as models for classes that other classes should extend. Abstract classes, of course, cannot be themselves instantiated, because it is nonsensical to do so. If you wish, you can delve deeper into the details of Java Swing and the other classes used in this lab.

Zip your lab and submit it to Canvas. Let your graders know, then you may leave.