Due Date: 16.6.24

Collision Detection, Paddle and Blocks

In the previous assignment, you were introduced to the biuoop classes. You created a Ball object that bounces around the screen, changing direction as it hits the boundaries.

The Ball is the first component of our Arkanoid game. In this assignment you will build two additional components: `Block`, which are obstacles that appear on the screen, and `Paddle` which represents the "player" & is controlled by the keyboard. When the ball hits either the a block or the paddle, it will change its direction. In this assignment, the blocks will remain on the screen after being hit -- we will deal with block removal in future assignments.

A new Object-oriented construct we will use are Interfaces.

In this assignment, you will create many classes/methods and the project is starting to grow. It's a good time to apply what we learnt in class about testing your code, the earlier the better.

We supply the build.xml for the assignment.

Interfaces

Very briefly, an interface is a contract between a class and a user.

When we say: "The interface Y has methods A,B,C" it means that the name Y defines a list of methods and their signatures (parameter types and return types).

When we say: "class X implements interface Y" it means that the class must have the methods defined by interface Y (in our case, the methods A,B,C). The class must also declare that it is implementing the interface.

For example:

```
public interface Y {
   void A();
   int B(int i1, int i2);
   void C(int i3);
}

public class X implements Y {
    // members
    // constructor
    public X() {
        // ....
    }
    // methods
    // ...
    void A() {
        // ...
    }
    int B(int i1, int i2) {
        // ...
    }
    void C(int i3) {
        // ...
    }
    // it is OK to have additional methods not specified in the interface.
}
```

So, when we say "class X should implement interface Y", you can think of it as saying: "here is list of methods that class X must have".

biuoop

We remind you that the biuoop package and documentation are available here.

Part 1 -- Collision Detection

In the previous assignment, your Ball bounced whenever it crossed the screen boundary. In this assignment, it should bounce whenever it hits a Collidable object, such as a Block or a Paddle (or the edges of the screen). In order to identify collisions, we will need a simple **collision-detection** algorithm.

A Rectangle Class

The first step would be to create a Rectangle class, in addition to the Point and Line classes you already have. Our Rectangles will all be aligned with the axes.

The Rectangle class should have **at least** the following methods (you can add more, of course):

```
public class Rectangle {

   // Create a new rectangle with location and width/height.
   public Rectangle(Point upperLeft, double width, double height);

   // Return a (possibly empty) List of intersection points
   // with the specified line.
   public java.util.List<Point> intersectionPoints(Line line);

   // Return the width and height of the rectangle
   public double getWidth();
   public double getHeight();

   // Returns the upper-left point of the rectangle.
   public Point getUpperLeft();
}
```

In the intersectionPoints method you need to return a List.

In addition, add the following method to Line:

```java
public class Line {

   // If this line does not intersect with the rectangle, return null.
   // Otherwise, return the closest intersection point to the
   // start of the line.
   public Point closestIntersectionToStartOfLine(Rectangle rect)
}
```

When creating `Rectangle` and implementing its method, keep in mind that you can use functionality created in the previous assignment to make your life easier.

The Collidable interface

The `Collidable` interface will be used by things that can be collided with. In this assignment, this means the `Block`s and the `Paddle` (in order for the ball to bounce from the edges of the screen, we will place large blocks there). What do we need to know about things we may collide with? First, we need to know their location and size (to know if we are colliding with it or not). Second, we need to know what happens when the collision occurs. In our world, everything that we can collide into is rectangular. Thus, the location and size will be specified using a Rectangle. The Collidable interface is the following:

```java


public interface Collidable {
   // Return the "collision shape" of the object.
   Rectangle getCollisionRectangle();

   // Notify the object that we collided with it at collisionPoint with
   // a given velocity.
   // The return is the new velocity expected after the hit (based on
   // the force the object inflicted on us).
   Velocity hit(Point collisionPoint, Velocity currentVelocity);
}
```

So we have an interface, but we cannot create objects out of it! We know that `Block` is going to be something we collide into, so let's start there. This will be a good time to implement a `Block` class implementing the `Collidable` interface. For now, it is enough to implement only these methods, we will add more methods later.

Reminder, in the previous assignment, when the ball hits the screen border, its velocity was updated according to the border type. A collision to the right/left border should change the horizontal direction and a collision to the top/bottom border should change the vertical direction of the velocity.

Now we have introduced the `Block` class, we don't have collision with the screen border anymore, rather to an instance of the `Block` class, that can be either a block in the middle of the game, or a block in the edge of the screen. However, the rule for changing the velocity remains the same, that is, if the ball hits a vertical edge of the block, the horizontal direction should change, and if the ball hits an horizontal edge of the block, the vertical direction should change.

GameEnvironment

During the game, there are going to be many objects a Ball can collide with. The `GameEnvironment` class will be a collection of such things. The ball will know the game environment, and will use it to check for collisions and direct its movement.

The GameEnvironment class should include the at least the following methods:

```
public class GameEnvironment {

   // add the given collidable to the environment.
   public void addCollidable(Collidable c);

   // Assume an object moving from line.start() to line.end().
   // If this object will not collide with any of the collidables
   // in this collection, return null. Else, return the information
   // about the closest collision that is going to occur.
   public CollisionInfo getClosestCollision(Line trajectory)

}
```

The `CollisionInfo` class should hold the following information:

```
public class CollisionInfo {
   // the point at which the collision occurs.
   public Point collisionPoint();

   // the collidable object involved in the collision.
   public Collidable collisionObject();
}
```

Finally, the Ball object should change such that its movement will be guided by the obstacles in the game environment:

1. Change the Ball object to hold a reference to a GameEnvironment.
2. The `moveOneStep()` method should change to support arbitrary collisions.

Here is the movement algorithm the ball should follow (we assume that a ball hits a surface if its center hits a surface, and we assume the ball will be small enough for this to look good):

```
1) compute the ball trajectory (the trajectory is "how the ball will move
 without any obstacles" -- its a line starting at current location, and
ending where the velocity will take the ball if no collisions will occur).

2) Check (using the game environment) if moving on this trajectory will
hit anything.

2.1) If no, then move the ball to the end of the trajectory.
```

```
2.2) Otherwise (there is a hit):
2.2.2) move the ball to "almost" the hit point, but just slightly before it.
2.2.3) notify the hit object (using its hit() method) that a collision occurred.
2.2.4) update the velocity to the new velocity returned by the hit() method.
```

Does it work?

This is a good location to set up a simple program to see if the collision mechanism and ball movement works! (of course, it would have been a good idea to perform some extra tests along the way as well...).

We suggest that before proceeding, you create a simple main that does the following:

- Create a `GUI`, a `Ball` and a few `Block`s.
- Add the blocks to a GameEnvironment (make sure the Ball knows about this environment)
- Create an animation loop that:
- draws the ball and the blocks to the screen (you may want to add a drawOn(DrawSurface d) method to Block, similar to the one you have for ball.
- move the ball

You may want to have some very wide or tall blocks at the borders of your screen so that the ball does not fall off the borders. Make sure things behave as expected -- the ball should bounce to the correct direction whenever it hits any of the blocks. The movement should look relatively smooth.

One thing to be particularly aware of is what happens when a ball hits a block: if it hits is from below or above, it should change the vertical direction. If it is hit from the sides, it should change its horizontal direction. Make sure this is indeed what happens!

Part 2 -- the Sprite interface

In computer graphics and games, a Sprite is a game object that can be drawn to the screen (and which is not just a background image).
Sprites can be drawn on the screen, and can be notified that time has passed (so that they know to change their position / shape / appearance / etc). In our design, all of the game objects (Ball, Block, Paddle, ...) are `Sprite`s -- they will implement the `Sprite` interface:

```
public interface Sprite {
   // draw the sprite to the screen
   void drawOn(DrawSurface d);
   // notify the sprite that time has passed
   void timePassed();
}
```

Notice that `Ball` and `Block` already implement a part of the interface -- they both have a `drawOn` method. In order to make it official, you need to add the "implements" part to the class definition, and implement also the `timePassed` method. What should `timePassed` do? For the ball, it should move one step. For the block, currently we do nothing. If we wanted to have animated blocks (for example, blocks that change their color over time, or have a different graphics effect) we could use the timePassed method to implement this behavior.

What do we gain by introducing the `Sprite` interface? We can now simplify the animation loop, and decouple it from the specific game objects -- the animation loop should not care that we have a ball and some blocks and a paddle and deal with each of them individually. As far as the animation loop is concerned, there is only a collection of sprites. It will then first draw all of the sprites, then notify all of them that time has passed. We can then dynamically add different sprites (say we want to suddenly have two balls instead of one, or introduce a new type of a block, or maybe a flying monster) without touching the logic of the animation loop -- as far as it concerned, nothing has changed: there's just a collection of sprites.

The new animation loop

Currently, the `GameEnvironment` holds a list of `Collidable` objects. Similarly, a `SpriteCollection` will hold a collection of sprites.

```
public class SpriteCollection {
   public void addSprite(Sprite s);

   // call timePassed() on all sprites.
   public void notifyAllTimePassed();

   // call drawOn(d) on all sprites.
   public void drawAllOn(DrawSurface d);
}
```

Please notice: the SpriteCollection class supports the addition of new sprites. Does it make sense then to store the sprites in a simple array? What else can you use?

Ok, so we have a collection of sprites. Where will it live? We will create a `Game` class that will hold the sprites and the collidables, and will be in charge of the animation (This class is currently fairly basic, we will change it in future assignments).

```
public class Game {
   private SpriteCollection sprites;
   private GameEnvironment environment;

   public void addCollidable(Collidable c);
   public void addSprite(Sprite s);

   // Initialize a new game: create the Blocks and Ball (and Paddle)
   // and add them to the game.
   public void initialize();

   // Run the game -- start the animation loop.
```

```
    public void run();
}
```

The `initialize()` method is in charge of setting up the game -- creating the GUI, adding the blocks in a nice pattern, adding the ball (associating the ball with the GameEnvironment) and adding any other game objects (such as the paddle). In this assignment, the size of the GUI is 800 (width) * 600 (height). The `run()` method is the animation loop, that will go over all the sprites, and call drawOn and timePassed on each of them.

The `run()` method will be the similar to the game loop from before, but will use the sprites collection:

```
public void run() {
    //...
    int framesPerSecond = 60;
    int millisecondsPerFrame = 1000 / framesPerSecond;
    while (true) {
        long startTime = System.currentTimeMillis(); // timing

        DrawSurface d = gui.getDrawSurface();
        this.sprites.drawAllOn(d);
        gui.show(d);
        this.sprites.notifyAllTimePassed();

        // timing
        long usedTime = System.currentTimeMillis() - startTime;
        long milliSecondLeftToSleep = millisecondsPerFrame - usedTime;
        if (milliSecondLeftToSleep > 0) {
            sleeper.sleepFor(milliSecondLeftToSleep);
        }
    }
}
```

First, above the loop we set the `framePerSeconds` number to 60 (we want a smooth animations that displays 60 different frames in a second, if possible). This means that each frame in the animation can last `1000 / framesPerSecond` milliseconds (we keep this in `millisecondsPerFrame`).

Another change from the previous loop are the lines marked as `// timing`. Because we are now doing more work in the loop body, the time it takes to perform the work may be non-negligible. We therefor subtract the time it takes to do the work from the sleep time of `millisecondsPerFrame` milliseconds.

Now let's discuss `initialize`. Note that a Ball is just a sprite, but a Block is both a sprite and a collidable. So when we add a Ball we need to call only `addSprite()` but when we add a Block we need to call both `addSprite` and `addCollidable`. What if we forget one of them? Or if we try to add ball as a collidable? This is error-prone. What can we do? One way around this is to let each of the game objects know how they should be added to the game. In this example, both Ball and Sprite will have a `addToGame(Game g)` method. This method will be in charge of adding the ball and the block to the game, calling the appropriate game methods.

So, the body of `initialize()` will look something like this (not only):

```
public class Game {
   public void initialize() {
      Ball ball = new Ball(...);
      b.addToGame(this);
      for (...) {
         Block block = new Block(...);
         block.addToGame(this);
      }
   }
}
```

Make it work

Create a program with main that does the following, and make sure that it works (that is, creates a ball and blocks, and the ball is bouncing around the screen hitting blocks and changing directions).

```
public static void main(String[] args) {
   Game game = new Game();
   game.initialize();
   game.run();
}
```

Part 3 -- the Paddle

The `Paddle` is the player in the game. It is a rectangle that is controlled by the arrow keys, and moves according to the player key presses. It should implement the Sprite and the Collidable interfaces. It should also know how to move to the left and to the right:

```
public class Paddle implements Sprite, Collidable {
   private biuoop.KeyboardSensor keyboard;

   public void moveLeft();
   public void moveRight();

   // Sprite
   public void timePassed();
   public void drawOn(DrawSurface d);

   // Collidable
   public Rectangle getCollisionRectangle();
   public Velocity hit(Point collisionPoint, Velocity currentVelocity);

   // Add this paddle to the game.
   public void addToGame(Game g);
}
```

How does the Paddle move? its `timePassed` method should check if the "left" or "right" keys are pressed, and if so move it accordingly.

The paddle moves in a **circular fashion**, so that when it reaches the edge of the screen it moves to the other side.

Reading key presses

In order to read the key presses, you can use the `KeyboardSensor` interface from the biuoop class. A `KeyboardSensor` is created from a gui object:

```
//...
biuoop.GUI gui = new biuoop.GUI(...);
biuoop.KeyboardSensor keyboard = gui.getKeyboardSensor();
//...
```

Once created, you can ask it if a key is currently pressed:

```
if (keyboard.isPressed("a")) {
  System.out.println("the 'a' key is pressed");
}
```

In order to check for "special" keys such as the arrows or the return key, use the special constants defined in KeyboardSensor, for example:

```
if (keyboard.isPressed(KeyboardSensor.LEFT_KEY)) {
  System.out.println("the 'left arrow' key is pressed");
}
```

A more "Fun" Paddle:

In order for the game to be more enjoyable, we require the following behavior from the paddle: think of the paddle as having 5 equally-spaced regions. The behavior of the ball's bounce depends on where it hits the paddle. Let's denote the left-most region as 1, the rightmost as 5 & everything in between accordingly (for example the middle region is 3). If the ball hits the middle region (region 3), it should keep its horizontal direction and only change its vertical one (like when hitting a block). However, if we hit region 1, the ball should bounce back with an angle of 300 degrees (-60), regardless of where it came from. Remember, angle 0 = 360 is "up", so 300 means "a lot to the left". Similarly, for region 2 is should bounce back 330 degrees (a little to the left), for region 4 it should bounce in 30 degrees, and for region 5 in 60 degrees.

Make it work

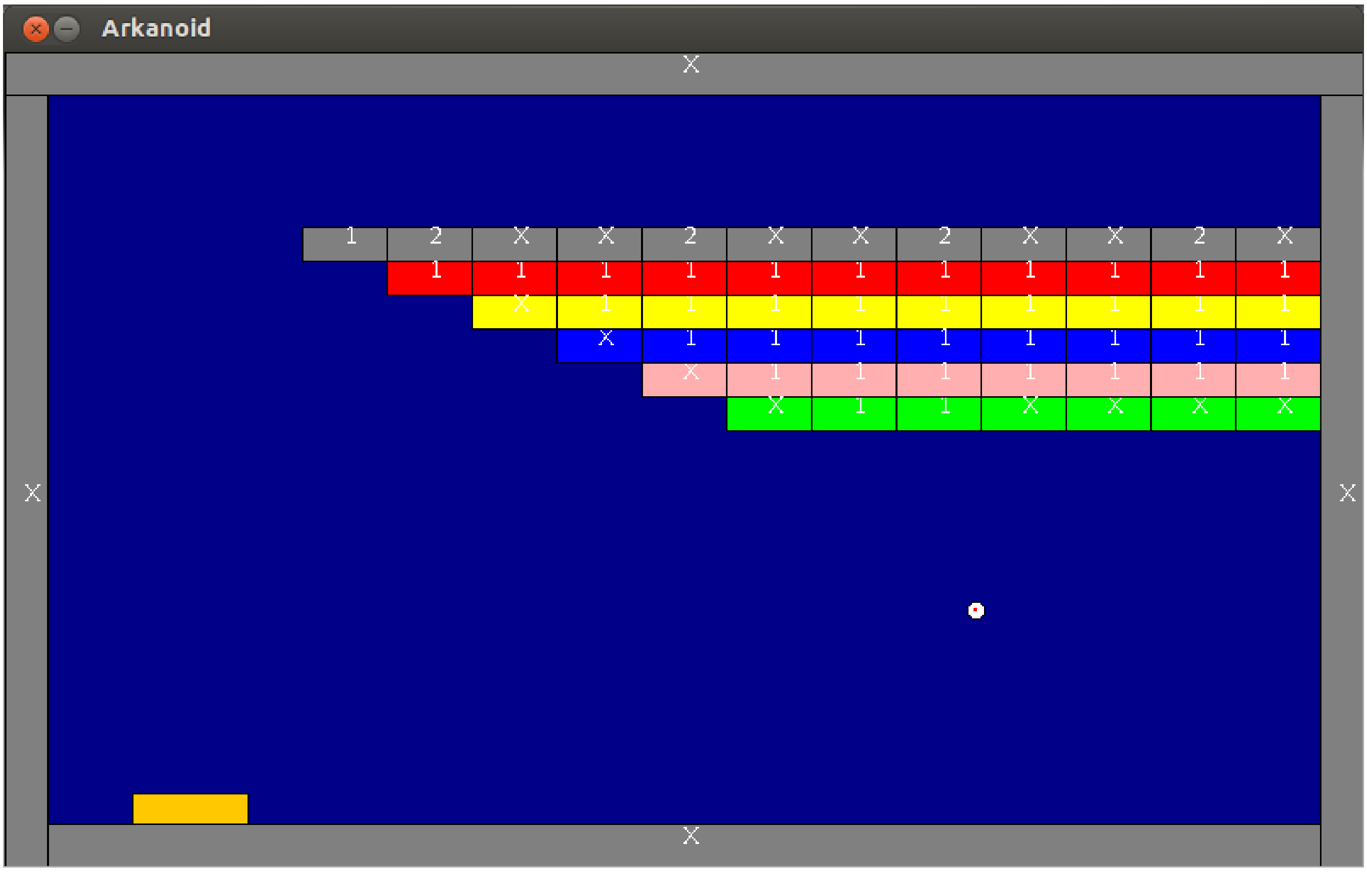
The paddle should hold a KeyboardSensor object, which should be passed to it upon construction.

Update the Game object to include also a user-controlled paddle, and make sure the ball bounces when it hits the paddle.

Part 4 -- putting it all together

Create a class file called `Ass3Game` with a main method that does the following:

- Create a game object, initializes and runs it. Reminder, the screen needs to be 800 * 600.
- The game should include a paddle (which is controlled by the left and right arrows), **two** balls, and blocks.
- The blocks should be arranged in a pattern following the image below. The exact sizes and colors are not important, but choose reasonable sizes and make each row a different color. Also, ignore the numbers and `X` in the blocks.
- The balls should never leave the screen, it's your responsibility to ensure this never happens. Hint: there is a simple way to do this.



What to submit:

Your submission should include all the classes and interfaces described above. Since this assigment is based on the previous one, you also need to submit the classes like `Point`, `Line`, etc you developed in `Ass2`. On the other hand, don't submit classes unrelated to this assignment such as `MultipleBouncingBallsAnimation`.

Several End Notes

- Remember, the paddle moves in a circular fashion, so that when it reaches the edge of the screen it moves to the other side.

- All the interfaces, classes, methods need to be implemented without changing the name or the signature of the method.
- You are allowed to add fields, methods, classes, interfaces to your project.
- You are allowed to use all the packages of the standard Java library except the `java.awt.geom.Line2D` class.
- You are not allowed to use the function `instanceof`.

Like in the previous assignment, it is OK to have the `hashCode` checkstyle error.