```java
import animation.AnimationRunner;
import biuoop.GUI;
import game.GameFlow;
import game.HighScoresTable;
import animation.KeyPressStoppableAnimation;
import animation.Task;
import animation.MenuAnimation;
import animation.Menu;
import animation.HighScoresAnimation;
import levels.LevelSpecificationReader;
import game.ShowHiScoresTask;
import levels.LevelInformation;

import java.io.BufferedReader;
import java.io.File;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.List;

/**
 * Classname: Ass6Game
 * Creates a new arkanoid game, initialize and runs the game.
 *
 * @author Elad Israel
 * @version 4.0 16/06/2018
 */
public class Ass6Game {
    /**
     * When runs, it creates a new game and runs the game.
     *
     * @param args not used
     */
    public static void main(String[] args) {
        final int frameWidth = 800;
        final int frameHeight = 600;
        GUI gui = new GUI("Arkanoid Game", frameWidth, frameHeight);
        AnimationRunner animationRunner = new AnimationRunner(gui);

        //loading or creating a highscore file.
        File highscoresFile = new File("highscores");
        HighScoresTable highScoresTable = new HighScoresTable(5);
        if (!highscoresFile.exists()) {
            try {
                highScoresTable.save(highscoresFile);
            } catch (IOException e) {
                System.err.println("Failed saving file");
            }
        } else {
            try {
                highScoresTable.load(highscoresFile);
            } catch (IOException e) {
                System.err.println("Failed loading file");
            }
        }

        Menu<Task<Void>> menu = new MenuAnimation<Task<Void>>("Arkanoid", gui.getKeyboardSensor(), animationRunner);
        KeyPressStoppableAnimation highscoresAnimation = new KeyPressStoppableAnimation(gui.getKeyboardSensor(),
                "space", new HighScoresAnimation(highScoresTable, gui.getKeyboardSensor()));

        Menu<Task<Void>> subMenu = new MenuAnimation<Task<Void>>("Levels Sets", gui.getKeyboardSensor(),
                animationRunner);
        String levelSetsPath;
        if (args.length == 1) {
            levelSetsPath = args[0];
        } else if (args.length == 0) {
            levelSetsPath = "level_sets.txt";
        } else {
            throw new RuntimeException("invalid parameters");
        }
        BufferedReader levelSetsReader;
        try {
            levelSetsReader = new BufferedReader(new
                    InputStreamReader(ClassLoader.getSystemClassLoader().getResourceAsStream(levelSetsPath)));

        } catch (Exception e) {
            throw new RuntimeException("couldn't load level sets file");
        }
        String line;
        String levelKey;
        String levelMessage;
        try {
            do {
                line = levelSetsReader.readLine();
                if (line == null) {
                    break;
                }
                levelKey = line.substring(0, 1);
                if (!line.substring(1, 2).equals(":")) {
                    throw new Exception("invalid levelSet format");
                }
                levelMessage = line.substring(2);

                line = levelSetsReader.readLine();
                if (line == null) {
                    throw new Exception("level name without level path");
                }
                final String levelDefPath = line;
                subMenu.addSelection(levelKey, levelMessage, new Task<Void>() {
                    @Override
                    public Void run() {
                        GameFlow gameFlow = new GameFlow(animationRunner, gui.getKeyboardSensor(), frameWidth,
                                frameHeight, 7, highScoresTable);
                        BufferedReader reader = new BufferedReader(new InputStreamReader(ClassLoader
                                .getSystemClassLoader().getResourceAsStream(levelDefPath)));

                        LevelSpecificationReader levelSpecReader = new LevelSpecificationReader();
                        List<LevelInformation> levelsInformation = levelSpecReader.fromReader(reader);
                        gameFlow.runLevels(levelsInformation);
                        return null;
                    }
                });
            } while (true);
        } catch (Exception e) {
            throw new RuntimeException("reading Blocks definition failed!");
        }

        menu.addSubMenu("s", "Game", subMenu);

        menu.addSelection("h", "Hi scores", new ShowHiScoresTask(animationRunner, highscoresAnimation));

        menu.addSelection("q", "Quit", new Task<Void>() {
            @Override
            public Void run() {
                System.exit(0);
                return null;
            }
        });


        while (true) {
            animationRunner.run(menu);
            // wait for user selection
            Task<Void> task = menu.getStatus();
            task.run();
        }
    }
}
```

```
package game;

import animation.GameLevel;
import biuoop.DrawSurface;
import levels.Background;
import listeners.HitListener;
import listeners.HitNotifier;
import shapes.Ball;
import shapes.Point;
import shapes.Rectangle;

import java.awt.Color;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * Classname: Block
 * Blocks are obstacles on the screen.
 * a Block (actually, a Rectangle) has size (as a rectangle), color, and location (a Point).
 * Blocks also know how to draw themselves on a DrawSurface.
 * A block can also notify the object that we collided with it about the new velocity it should have after collision.
 *
 * @author Elad Israel
 * @version 4.0 17/06/2018
 */
public class Block implements Collidable, Sprite, HitNotifier {

    private Rectangle collosionRectangle;
    private int hitPoints;
    private List<HitListener> hitListeners;
    private Map<Integer, Background> hpToBackground;
    private Background background;


    /**
     * Instantiates a new Block.
     *
     * @param rectangle the rectangle
     */
    public Block(Rectangle rectangle) {
        this.background = null;
        this.hpToBackground = new HashMap<>();
        this.collosionRectangle = rectangle;
    }

    /**
     * Constructor1
     * construct a Block using upper-left X coordinate, upper-left Y coordinate, width and height.
     *
     * @param upperLeftX upper-left corner X coordinate
     * @param upperLeftY upper-left corner Y coordinate
     * @param width      of the rectangle
     * @param height     of the rectangle
     */
    public Block(double upperLeftX, double upperLeftY, double width, double height) {
        this(new Rectangle(new Point(upperLeftX, upperLeftY), width, height));
    }

    /**
     * Constructor2
     * construct a Block using upper-left X coordinate, upper-left Y coordinate, width, height and color to fill.
     *
     * @param upperLeftX upper-left corner X coordinate
     * @param upperLeftY upper-left corner Y coordinate
     * @param width      of the rectangle
     * @param height     of the rectangle
     * @param fillColor  of the rectangle
     */
    public Block(double upperLeftX, double upperLeftY, double width, double height, java.awt.Color fillColor) {
        this(new Rectangle(new Point(upperLeftX, upperLeftY), width, height, fillColor));
    }

    /**
     * Constructor2
     * construct a Block using upper-left X coordinate, upper-left Y coordinate,
     * width, height and colors to fill and draw.
     *
     * @param upperLeftX upper-left corner X coordinate
     * @param upperLeftY upper-left corner Y coordinate
     * @param width      of the rectangle
     * @param height     of the rectangle
     * @param fillColor  of the rectangle
     * @param drawColor  of the rectangle
     */
    public Block(double upperLeftX, double upperLeftY, double width, double height, java.awt.Color fillColor, java.awt
            .Color drawColor) {
        this(new Rectangle(new Point(upperLeftX, upperLeftY), width, height, fillColor, drawColor));
    }
```

```java
    /**
     * Constructor2
     * construct a Block using upper-left X coordinate, upper-left Y coordinate,
     * width, height and colors to fill and draw.
     *
     * @param upperLeftX upper-left corner X coordinate
     * @param upperLeftY upper-left corner Y coordinate
     * @param width      of the rectangle
     * @param height     of the rectangle
     * @param fillColor  of the rectangle
     * @param drawColor  of the rectangle
     * @param hitPoints  of the rectangle
     */
    public Block(double upperLeftX, double upperLeftY, double width, double height, java.awt.Color fillColor, java.awt
            .Color drawColor, int hitPoints) {
        this(new Rectangle(new Point(upperLeftX, upperLeftY), width, height, fillColor, drawColor));
        this.hitPoints = hitPoints;
    }

    /**
     * Sets collosion rectangle.
     *
     * @param rectangle the collosion rectangle
     */
    public void setCollosionRectangle(Rectangle rectangle) {
        this.collosionRectangle = rectangle;
    }

    /**
     * Sets background.
     *
     * @param backgroundToSet the background
     */
    public void setBackground(Background backgroundToSet) {
        this.background = backgroundToSet;
    }

    /**
     * Sets hp to background.
     *
     * @param hpToBackgroundToSet the hp to background
     */
    public void setHpToBackground(Map<Integer, Background> hpToBackgroundToSet) {
        this.hpToBackground = hpToBackgroundToSet;
    }

    /**
     * Sets stroke.
     *
     * @param stroke the stroke
     */
    public void setStroke(Color stroke) {
        this.getCollisionRectangle().setDrawColor(stroke);
    }

    /**
     * Sets width.
     *
     * @param width the width
     */
    public void setWidth(double width) {
        this.collosionRectangle.setWidth(width);
    }

    /**
     * Sets height.
     *
     * @param height the height
     */
    public void setHeight(double height) {
        this.collosionRectangle.setHeight(height);
    }

    /**
     * Add hp background.
     *
     * @param hp   the hit points
     * @param backgroundToSet the background
     */
    public void addHpBackground(int hp, Background backgroundToSet) {
        this.hpToBackground.put(hp, backgroundToSet);
    }

    /**
     * Return the "collision shape" of the object - the rectangle.
     *
     * @return collision shape- rectangle
     */
    public Rectangle getCollisionRectangle() {
        return collosionRectangle;
    }

    /**
     * draws this block on the given DrawSurface.
     * also, draws its hitPoints.
     *
     * @param surface drawSurface
     */
    public void drawOn(DrawSurface surface) {

        if (this.hpToBackground.containsKey(this.hitPoints)) {
            Background backgroundFromHp = this.hpToBackground.get(this.hitPoints);
            backgroundFromHp.drawOn(surface, collosionRectangle);
            if (this.collosionRectangle.getDrawColor() != null) {
                surface.setColor(this.collosionRectangle.getDrawColor());
                surface.drawRectangle((int) this.collosionRectangle.getUpperLeft().getX(), (int) this
                                .collosionRectangle.getUpperLeft().getY(),
                        (int) this.collosionRectangle.getWidth(), (int) this.collosionRectangle.getHeight());
            }
        } else {
            if (this.background == null) {
                this.collosionRectangle.drawOn(surface);
            } else { //background!=null
                this.background.drawOn(surface, collosionRectangle);
                if (this.collosionRectangle.getDrawColor() != null) {
                    surface.setColor(this.collosionRectangle.getDrawColor());
                    surface.drawRectangle((int) this.collosionRectangle.getUpperLeft().getX(), (int) this
                                    .collosionRectangle.getUpperLeft().getY(),
                            (int) this.collosionRectangle.getWidth(), (int) this.collosionRectangle.getHeight());
                }
            }
        }
    }

    /**
     * Gets hit points.
     *
     * @return the hit points
     */
    public int getHitPoints() {
        return this.hitPoints;
    }

    /**
     * Sets hit points.
     *
     * @param hp the hit points
     */
    public void setHitPoints(int hp) {
        this.hitPoints = hp;
    }

    /**
     * Specify what the block does when time is passed. (currently- nothing).
     * @param dt amount of seconds passed since the last call
     */
    public void timePassed(double dt) {

    }

    /**
     * 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).
     *
     * @param collisionPoint  the point of collision
     * @param currentVelocity the velocity of the ball before impact.
     * @param hitter          the ball that hits the block.
     * @return the new velocity the ball should have after the collision.
     */
    public Velocity hit(Ball hitter, Point collisionPoint, Velocity currentVelocity) {
        this.notifyHit(hitter);
        //lower the block's HP
        if (this.hitPoints > 0) {
            this.hitPoints--;
        }
        //checks on which edge the collision point is and returns the appropriate velocity accordingly.
        if (collosionRectangle.getUpperEdge().isPointOnTheLine(collisionPoint)) {
            return new Velocity(currentVelocity.getDx(), -1 * currentVelocity.getDy());
        }
        if (collosionRectangle.getLowerEdge().isPointOnTheLine(collisionPoint)) {
            return new Velocity(currentVelocity.getDx(), -1 * currentVelocity.getDy());
        }
        if (collosionRectangle.getLeftEdge().isPointOnTheLine(collisionPoint)) {
            return new Velocity(-1 * currentVelocity.getDx(), currentVelocity.getDy());
        }
        if (collosionRectangle.getRightEdge().isPointOnTheLine(collisionPoint)) {
            return new Velocity(-1 * currentVelocity.getDx(), currentVelocity.getDy());
        }
        return currentVelocity;
    }

    /**
     * will be called whenever a hit() occurs,
     * and notifiers all of the registered HitListener objects by calling their hitEvent method.
     *
     * @param hitter the ball that hit the block.
     */
    private void notifyHit(Ball hitter) {
        if (this.hitListeners == null) {
            this.hitListeners = new ArrayList<>();
        }
        // Make a copy of the hitListeners before iterating over them.
        List<HitListener> listeners = new ArrayList<HitListener>(this.hitListeners);
        // Notify all listeners about a hit event:
        for (HitListener hl : listeners) {
            hl.hitEvent(this, hitter);
        }
    }

    /**
     * adds the block to the game-as a sprite and as a Collidable.
     * also, increases the number of blocks in the game.
     *
     * @param g game
     */
    public void addToGame(GameLevel g) {
        g.addSprite(this);
        g.addCollidable(this);
        g.getNumOfBlocks().increase(1);
    }

    /**
     * removes the block from the gameLevel-as a sprite and as a Collidable.
     * Decrease in numOfBalls is executed in BallsRemover.
     *
     * @param gameLevel gameLevel
     */
    public void removeFromGame(GameLevel gameLevel) {
        gameLevel.removeSprite(this);
        gameLevel.removeCollidable(this);
    }

    /**
     * Add hl as a listener to hit events.
     *
     * @param hl HitListener to remove
     */
    public void addHitListener(HitListener hl) {
        if (this.hitListeners == null) {
            this.hitListeners = new ArrayList<>();
        }
        this.hitListeners.add(hl);
    }

    /**
     * Remove hl from the list of listeners to hit events.
     *
     * @param hl HitListener to remove
     */
    public void removeHitListener(HitListener hl) {
        if (this.hitListeners == null) {
            throw new RuntimeException("hitListeners List wasn't initialized."
                    + "cannot remove listener if no listeners were added");
        }
        this.hitListeners.remove(hl);
    }
}
```

```java
package game;

import animation.GameLevel;
import biuoop.DrawSurface;
import biuoop.KeyboardSensor;
import shapes.Ball;
import shapes.Line;
import shapes.Point;
import shapes.Rectangle;

/**
 * ClassName: 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 implements the Sprite and the Collidable interfaces.
 * It  also knows how to move to the left and to the right.
 *
 * @author Elad Israel
 * @version 3.0 20/05/2018
 */
public class Paddle implements Sprite, Collidable {

    private static final int SIDE_FRAMES_WIDTH = 25;
    private static final int FRAME_WIDTH = 800;
    private double speed;
    private Rectangle paddle;
    private KeyboardSensor keyboardSensor;


    /**
     * Construct the Paddle using position point, width, height, fill color, draw color.
     * also receives the keyboardSensor and sets it.
     *
     * @param upperLeft      point(position)
     * @param width          of the paddle
     * @param height         of the paddle
     * @param speed          of the paddle
     * @param fillColor      of the paddle
     * @param drawColor      of the paddle
     * @param keyboardSensor passed in order to identify the movements of the Paddle.
     */
    public Paddle(Point upperLeft, double width, double height, double speed, java.awt.Color fillColor, java.awt.Color
            drawColor, KeyboardSensor keyboardSensor) {
        this.paddle = new Rectangle(upperLeft, width, height, fillColor, drawColor);
        this.keyboardSensor = keyboardSensor;
        this.speed = speed;
    }

    /**
     * moves the paddle to the left.
     *
     * @param dt amount of seconds passed since the last call
     */
    public void moveLeft(double dt) {
        //reached left edge(block)
        if (paddle.getUpperLeft().getX() - speed * dt <= SIDE_FRAMES_WIDTH) {
            return;
        }
        paddle.changePosition(new Point(paddle.getUpperLeft().getX() - speed * dt, paddle.getUpperLeft().getY()));
    }

    /**
     * moves the paddle to the right.
     *
     * @param dt amount of seconds passed since the last call
     */
    public void moveRight(double dt) {
        //reached right edge(block)
        if (paddle.getUpperLeft().getX() + paddle.getWidth() + speed * dt >= FRAME_WIDTH - SIDE_FRAMES_WIDTH) {
            return;
        }
        paddle.changePosition(new Point(paddle.getUpperLeft().getX() + speed * dt, paddle.getUpperLeft().getY()));
    }

    /**
     * Specify what the paddle does when time is passed - moves left or right if pressed.
     *
     * @param dt amount of seconds passed since the last call
     */
    public void timePassed(double dt) {
        if (keyboardSensor.isPressed(keyboardSensor.LEFT_KEY)) {
            moveLeft(dt);
        }
        if (keyboardSensor.isPressed(keyboardSensor.RIGHT_KEY)) {
            moveRight(dt);
        }
    }

    /**
     * draws the paddle on the surface.
     *
     * @param d draw surface
     */
    public void drawOn(DrawSurface d) {
        this.paddle.drawOn(d);
    }

    /**
     * Return the "collision shape" of the object.
     *
     * @return collision shape- rectangle
     */
    public Rectangle getCollisionRectangle() {
        return this.paddle;
    }

    /**
     * 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).
     * <p>
     * The paddle have 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 and the rightmost as 5 (so the middle region is 3).
     * If the ball hits the middle region (region 3), it keeps 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 it bounces back 330 degrees (a little to the left),
     * for region 4 it  bounces in 30 degrees, and for region 5 in 60 degrees.
     *
     * @param hitter          the ball that hit the paddle.
     * @param collisionPoint  the point of collision.
     * @param currentVelocity the velocity of the ball before impact.
     * @return the new velocity the ball should have after the collision.
     */
    public Velocity hit(Ball hitter, Point collisionPoint, Velocity currentVelocity) {
        Line upperEdge = this.paddle.getUpperEdge();
        // divides the paddle's upper edge to 5 equally-spaced regions
        double upperEdgeRegionLength = upperEdge.length() / 5;
        //calculates the speed using Pythagoras (sqrt(dx^2+dy^2))=speed.
        double currentSpeed = Math.sqrt(Math.pow(currentVelocity.getDx(), 2) + Math.pow(currentVelocity.getDy(), 2));

        // calculates the 5 regions
        Line leftMostRegion = new Line(upperEdge.start(), new Point(upperEdge.start().getX() + upperEdgeRegionLength,
                upperEdge.start().getY()));
        Line leftMiddleRegion = new Line(new Point(upperEdge.start().getX() + upperEdgeRegionLength,
                upperEdge.start().getY()), new Point(upperEdge.start().getX() + 2 * upperEdgeRegionLength,
                upperEdge.start().getY()));
        Line middleRegion = new Line(new Point(upperEdge.start().getX() + 2 * upperEdgeRegionLength,
                upperEdge.start().getY()), new Point(upperEdge.start().getX() + 3 * upperEdgeRegionLength,
                upperEdge.start().getY()));
        Line rightMiddleRegion = new Line(new Point(upperEdge.start().getX() + 3 * upperEdgeRegionLength,
                upperEdge.start().getY()), new Point(upperEdge.start().getX() + 4 * upperEdgeRegionLength,
                upperEdge.start().getY()));
        Line rightMostRegion = new Line(new Point(upperEdge.start().getX() + 4 * upperEdgeRegionLength,
                upperEdge.start().getY()), new Point(upperEdge.start().getX() + 5 * upperEdgeRegionLength,
                upperEdge.start().getY()));

        //deals with a collision according to the region(detailed explanation above)
        if (leftMostRegion.isPointOnTheLine(collisionPoint)) {
            return Velocity.fromAngleAndSpeed(300, currentSpeed);
        }
        if (leftMiddleRegion.isPointOnTheLine(collisionPoint)) {
            return Velocity.fromAngleAndSpeed(330, currentSpeed);
        }
        if (middleRegion.isPointOnTheLine(collisionPoint)) {
            return new Velocity(currentVelocity.getDx(), -1 * currentVelocity.getDy());
        }
        if (rightMiddleRegion.isPointOnTheLine(collisionPoint)) {
            return Velocity.fromAngleAndSpeed(30, currentSpeed);
        }
        if (rightMostRegion.isPointOnTheLine(collisionPoint)) {
            return Velocity.fromAngleAndSpeed(60, currentSpeed);
        }
        if (this.paddle.getLeftEdge().isPointOnTheLine(collisionPoint)) {
            return new Velocity(-1 * currentVelocity.getDx(), currentVelocity.getDy());
        }
        if (this.paddle.getRightEdge().isPointOnTheLine(collisionPoint)) {
            return new Velocity(-1 * currentVelocity.getDx(), currentVelocity.getDy());
        }
        return currentVelocity;
    }

    /**
     * adds the Paddle to the game-as a sprite and as a Collidable.
     *
     * @param g game
     */
    public void addToGame(GameLevel g) {
        g.addSprite(this);
        g.addCollidable(this);
    }

    /**
     * adds the Paddle to the game-as a sprite and as a Collidable.
     *
     * @param g game
     */
    public void removeFromGame(GameLevel g) {
        g.removeSprite(this);
        g.removeCollidable(this);
    }
}
```

```java
package game;

import biuoop.DrawSurface;

/**
 * interface name: Sprite
 * 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)
 *
 * @author Elad Israel
 * @version 3.0 20/05/2018
 */
public interface Sprite {
    /**
     * draw the sprite to the screen.
     *
     * @param d drawSurface
     */
    void drawOn(DrawSurface d);

    /**
     * notify the sprite that time has passed.
     *
     * @param dt amount of seconds passed since the last call
     */
    void timePassed(double dt);
}
```

```java
package game;

/**
 * Classname: Counter.
 * Counter is a simple class that is used for counting things.
 *
 * @author Elad Israel
 * @version 3.0 20/05/2018
 */
public class Counter {
    private int count;

    /**
     * Constructor - initialize the counter to 0.
     */
    public Counter() {
        this.count = 0;
    }

    /**
     * add number to current count.
     *
     * @param number increase by this number.
     */
    public void increase(int number) {
        this.count += number;
    }

    /**
     * subtract number from current count.
     *
     * @param number decrease by this number.
     */
    public void decrease(int number) {
        this.count -= number;
    }

    /**
     * get current count.
     *
     * @return the value of the count
     */
    public int getValue() {
        return this.count;
    }
}
```

```java
package game;

import animation.EndScreen;
import animation.HighScoresAnimation;
import animation.KeyPressStoppableAnimation;
import animation.GameLevel;
import animation.AnimationRunner;
import biuoop.DialogManager;
import biuoop.KeyboardSensor;
import levels.LevelInformation;
import java.io.File;
import java.io.IOException;
import java.util.List;

/**
 * interface name: GameFlow
 * In charge of creating the different levels, and moving from one level to the next.
 *
 * @author Elad Israel
 * @version 4.0 17/06/2018
 */
public class GameFlow {

    private final int frameWidth;
    private final int frameHeight;
    private AnimationRunner animationRunner;
    private KeyboardSensor keyboardSensor;
    private Counter score;
    private Counter numOfLives;
    private HighScoresTable highScoresTable;

    /**
     * Constructor.
     *
     * @param ar              the AnimationRunner
     * @param ks              the KeyboardSensor
     * @param frameWidth      the frame width
     * @param frameHeight     the frame height
     * @param numOfLives      the num of lives
     * @param highScoresTable the high scores table
     */
    public GameFlow(AnimationRunner ar, KeyboardSensor ks, final int frameWidth, final int frameHeight, int
            numOfLives, HighScoresTable highScoresTable) {
        this.animationRunner = ar;
        this.keyboardSensor = ks;
        this.score = new Counter();
        this.numOfLives = new Counter();
        this.numOfLives.increase(numOfLives);
        this.frameWidth = frameWidth;
        this.frameHeight = frameHeight;
        this.highScoresTable = highScoresTable;
    }


    /**
     * Run the specified levels on the list.
     *
     * @param levels the levels to run
     */
    public void runLevels(List<LevelInformation> levels) {
        for (LevelInformation levelInfo : levels) {

            GameLevel level = new GameLevel(levelInfo, this.keyboardSensor, this.animationRunner, this.score, this
                    .numOfLives, this.frameWidth, this.frameHeight);

            level.initialize();

            while (level.getBlocksLeftToRemove().getValue() > 0 && level.getNumOfLives().getValue() > 0) {
                level.playOneTurn();
            }

            if (level.getNumOfLives().getValue() <= 0) {
                this.animationRunner.run(new KeyPressStoppableAnimation(this.keyboardSensor, "space",
                        new EndScreen(this.keyboardSensor, this.score, false)));
                break;
            }
        }
        //end of game run
        if (this.numOfLives.getValue() > 0) {
            this.animationRunner.run(new KeyPressStoppableAnimation(this.keyboardSensor, "space", new EndScreen(this
                    .keyboardSensor, this.score, true)));

        }

        File highscoresFile = new File("highscores");
        if (highScoresTable.isHighScore(this.score)) {
            DialogManager dialog = this.animationRunner.getGui().getDialogManager();
            String name = dialog.showQuestionDialog("Name", "What is your name?", "");
            highScoresTable.add(new ScoreInfo(name, this.score.getValue()));
        }
        try {
            highScoresTable.save(highscoresFile);
        } catch (IOException e) {
            System.err.println("Failed saving file");
        }
        this.animationRunner.run(new KeyPressStoppableAnimation(this.keyboardSensor, "space",
                new HighScoresAnimation(highScoresTable, this.keyboardSensor)));
    }
}
```

```java
package game;

import shapes.Point;

/**
 * Classname: Velocity
 * Velocity specifies the change in position on the `x` and the `y` axes.
 *
 * @author Elad Israel
 * @version 3.0 20/05/2018
 */
public class Velocity {

    //members
    private double dx;
    private double dy;

    /**
     * Constructor1.
     * Constructs a Velocity using dx(change in X-coordinate) and dy(change in Y-coordinate).
     *
     * @param dx change in X-coordinate.
     * @param dy change in Y-coordinate.
     */
    public Velocity(double dx, double dy) {
        this.dx = dx;
        this.dy = dy;
    }

    /**
     * Constructor2.
     * Constructs a Velocity using another Velocity.
     *
     * @param velocity - the new Velocity to set.
     */
    public Velocity(Velocity velocity) {
        this.dx = velocity.getDx();
        this.dy = velocity.getDy();
    }

    /**
     * Constructor3.
     * Constructs a Velocity using an angle and speed.
     *
     * @param angle angle of movement.
     * @param speed speed of movement.
     * @return new velocity from the calculated dx and dy.
     */
    public static Velocity fromAngleAndSpeed(double angle, double speed) {
        //change in X-coordinate=speed*sin(rad) of the angle
        double dx = speed * Math.sin(Math.toRadians(angle));
        //change in Y-coordinate=speed*cos(rad) of the angle. multiply by -1 to fix and reverse the axes (upside-down).
        double dy = -1 * speed * Math.cos(Math.toRadians(angle));
        return new Velocity(dx, dy);
    }

    /**
     * Access method- Return the dx value of this point.
     *
     * @return dx value of this point
     */
    public double getDx() {
        return this.dx;
    }

    /**
     * sets the dx value of this point.
     *
     * @param dX value to set to this point
     */
    public void setDx(double dX) {
        this.dx = dX;
    }

    /**
     * Access method- Return the dy value of this point.
     *
     * @return dy value of this point
     */
    public double getDy() {
        return this.dy;
    }

    /**
     * sets the dy value of this point.
     *
     * @param dY value to set to this point
     */
    public void setDy(double dY) {
        this.dy = dY;
    }

    /**
     * Takes a point with position (x,y) and return a new point with position (x+dx, y+dy)
     * Actually changing the objects position.
     *
     * @param p point given
     * @param dt amount of seconds passed since the last call
     * @return new point with updated location
     */
    public Point applyToPoint(Point p, double dt) {
        return new Point(p.getX() + dx * dt, p.getY() + dy * dt);
    }
}
```

```java
package game;

import java.io.Serializable;

/**
 * Contains the information about the score.
 *
 * @author Elad Israel
 * @version 4.0 17/06/2018
 */
public class ScoreInfo implements Serializable {

    private String name;
    private int score;

    /**
     * Instantiates a new Score info.
     *
     * @param name  the name
     * @param score the score
     */
    public ScoreInfo(String name, int score) {
        this.name = name;
        this.score = score;
    }

    /**
     * Gets name.
     *
     * @return the name
     */
    public String getName() {
        return this.name;
    }

    /**
     * Gets score.
     *
     * @return the score
     */
    public int getScore() {
        return this.score;
    }
}
```

```java
package game;

import shapes.Ball;
import shapes.Point;
import shapes.Rectangle;

/**
 * interface name: Collidable
 * The game.Collidable interface is used by things that can be collided with.
 * A collidable object must have location and size(collision rectangle)
 * and need to know what to do when a collision occurs.
 *
 * @author Elad Israel
 * @version 3.0 20/05/2018
 */
public interface Collidable {
    /**
     * Return the "collision shape" of the object.
     *
     * @return collision shape- rectangle
     */
    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).
     *
     * @param hitter          the ball that hit the coolidable.
     * @param collisionPoint  the point of collision.
     * @param currentVelocity the velocity of the ball before impact.
     * @return the new velocity the ball should have after the collision.
     */
    Velocity hit(Ball hitter, Point collisionPoint, Velocity currentVelocity);
}
```

```java
package game;

import shapes.Point;

/**
 * Classname: CollisionInfo
 * Contains information about a collision- which object collided with any of the collidables
 * in this collection, and the closest collision point that is going to occur.
 *
 * @author Elad Israel
 * @version 3.0 20/05/2018
 */
public class CollisionInfo {

    private Collidable collidableObject;
    private Point collisionPoint;

    /**
     * Constructor.
     *
     * @param collidableObject the object that collided.
     * @param collisionPoint   the point of collision.
     */
    public CollisionInfo(Collidable collidableObject, Point collisionPoint) {
        this.collidableObject = collidableObject;
        this.collisionPoint = collisionPoint;
    }

    /**
     * Getter for collisionPoint.
     *
     * @return the point at which the collision occurs.
     */
    public Point collisionPoint() {
        return this.collisionPoint;
    }

    /**
     * Getter for collisionObject.
     *
     * @return the collidable object involved in the collision.
     */
    public Collidable collisionObject() {
        return this.collidableObject;
    }
}
```

```java
package game;

import animation.GameLevel;
import biuoop.DrawSurface;

import java.awt.Color;

/**
 * Class name: LivesIndicator
 * LivesIndicator sprite that will sit at the top of the screen and indicate the number of lives.
 *
 * @author Elad Israel
 * @version 3.0 20/05/2018
 */
public class LivesIndicator implements Sprite {
    private Counter livesCount;

    /**
     * Constructor.
     *
     * @param livesCount the lives count
     */
    public LivesIndicator(Counter livesCount) {
        this.livesCount = livesCount;
    }

    /**
     * Specify what the ScoreIndicator does when time is passed.
     * @param dt amount of seconds passed since the last call
     */
    public void timePassed(double dt) {

    }

    /**
     * draws the ScoreIndicator on the surface.
     *
     * @param d draw surface
     */
    public void drawOn(DrawSurface d) {
        final int lettersSize = 15;
        d.setColor(Color.black);
        d.drawText(100, 19, "Lives: " + this.livesCount.getValue(), lettersSize);
    }

    /**
     * adds the ScoreIndicator to the game-as a sprite.
     *
     * @param g game
     */
    public void addToGame(GameLevel g) {
        g.addSprite(this);
    }
}
```

```java
package game;

import animation.GameLevel;
import biuoop.DrawSurface;
import shapes.Point;
import shapes.Rectangle;

import java.awt.Color;

/**
 * Class name: ScoreIndicator
 * ScoreIndicator sprite that will sit at the top of the screen and indicate the score.
 *
 * @author Elad Israel
 * @version 3.0 20/05/2018
 */
public class ScoreIndicator implements Sprite {
    private Rectangle scoreRectangle;
    private Counter scoreCount;

    /**
     * Constructor.
     *
     * @param rectangle  the rectangle
     * @param scoreCount the score count
     */
    public ScoreIndicator(Rectangle rectangle, Counter scoreCount) {
        this.scoreRectangle = rectangle;
        this.scoreCount = scoreCount;
    }

    /**
     * Specify what the ScoreIndicator does when time is passed.
     *
     * @param dt amount of seconds passed since the last call
     */
    public void timePassed(double dt) {

    }

    /**
     * draws the ScoreIndicator on the surface.
     *
     * @param d draw surface
     */
    public void drawOn(DrawSurface d) {
        final int lettersSize = 15;
        Point upperLeft = this.scoreRectangle.getUpperLeft();
        double width = this.scoreRectangle.getWidth();
        double height = this.scoreRectangle.getHeight();
        this.scoreRectangle.drawOn(d);
        d.setColor(Color.black);
        d.drawText((int) (upperLeft.getX() + width / 2.2),
                (int) (upperLeft.getY() + height / 1.3), "Score: " + this.scoreCount.getValue(), lettersSize);
    }

    /**
     * adds the ScoreIndicator to the game-as a sprite.
     *
     * @param g game
     */
    public void addToGame(GameLevel g) {
        g.addSprite(this);
    }
}
```

```java
package game;

import shapes.Line;
import shapes.Point;

/**
 * interface name: GameEnvironment
 * A collection of the objects a Ball can collide with.
 * The ball will know the game environment, and will use it to check for collisions and direct its movement.
 *
 * @author Elad Israel
 * @version 3.0 20/05/2018
 */
public class GameEnvironment {

    private java.util.List<Collidable> collidableObjects;

    /**
     * Constructor.
     *
     * @param collidableObjects a collection of the objects a Ball can collide with.
     */
    public GameEnvironment(java.util.List<Collidable> collidableObjects) {
        this.collidableObjects = collidableObjects;
    }

    /**
     * add the given collidable to the environment.
     *
     * @param c given collidable
     */
    public void addCollidable(Collidable c) {
        collidableObjects.add(c);
    }

    /**
     * removes the given collidable from the environment.
     *
     * @param c given collidable
     */
    public void removeCollidable(Collidable c) {
        collidableObjects.remove(c);
    }


    /**
     * Getter for collidableObjects.
     *
     * @return a collection of the objects a Ball can collide with
     */
    public java.util.List<Collidable> getCollidable() {
        return collidableObjects;
    }


    /**
     * Assuming 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.
     *
     * @param trajectory a line representing the balls movement in the next step.
     * @return either null if no collision was found, or the CollisionInfo.
     */
    public CollisionInfo getClosestCollision(Line trajectory) {
        if (this.collidableObjects.isEmpty()) {
            return null;
        }
        //first intersection point
        Point firstInterP;
        //first intersection object
        Collidable firstInterO;
        //find index of the first collidable that intersects with trajectory
        int i = 0;
        while ((i < this.collidableObjects.size())
                && (trajectory.closestIntersectionToStartOfLine(collidableObjects.get(i).getCollisionRectangle())
                == null)) {
            i++;
        }
        //reached the end of the List and no collision was found with any of the Collidable objects
        if (i == this.collidableObjects.size()) {
            return null;
            //found first intersection with a Collidable. Doesn't have to be a collision yet(may not be the closest).
        } else {
            firstInterP = trajectory.closestIntersectionToStartOfLine(collidableObjects.get(i).getCollisionRectangle());
            firstInterO = this.collidableObjects.get(i);
        }
        //goes through all collidableObjects and finds all collisionPoints, and checks which one is the closest.
        Point closestInterP = firstInterP;
        Collidable closestInterO = firstInterO;
        for (; i < this.collidableObjects.size(); i++) {
            Point interPWithCurrentO = trajectory.closestIntersectionToStartOfLine(collidableObjects.get(i)
                    .getCollisionRectangle());
            //found another collisionPoint
            if (interPWithCurrentO != null) {
                //checks whether its the closest collisionPoint that was found(yet)
                if (interPWithCurrentO.distance(trajectory.start()) < closestInterP.distance(trajectory.start())) {
                    closestInterP = interPWithCurrentO;
                    closestInterO = this.collidableObjects.get(i);
                }
            }
        }
        return new CollisionInfo(closestInterO, closestInterP);
    }
}
```

```java
package game;


import java.io.File;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.Serializable;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.FileOutputStream;
import java.util.ArrayList;
import java.util.List;


/**
 * The High scores table.
 *
 * @author Elad Israel
 * @version 4.0 17/06/2018
 */
public class HighScoresTable implements Serializable {
    private List<ScoreInfo> scoresList;
    private int size;

    /**
     * Instantiates a new High scores table.
     * Create an empty high-scores table with the specified size.
     * The size means that the table holds up to size top scores.
     *
     * @param size the size
     */
    public HighScoresTable(int size) {
        scoresList = new ArrayList<>();
        this.size = size;
    }

    /**
     * Read a table from file and return it.
     * If the file does not exist, or there is a problem with
     * reading it, an empty table is returned.
     *
     * @param filename the filename
     * @return the high scores table
     */
    public static HighScoresTable loadFromFile(File filename) {
        //creates a new instance of highScoresTable
        HighScoresTable highScoresTable;
        ObjectInputStream objectInputStream = null;
        try {
            objectInputStream = new ObjectInputStream(new FileInputStream(filename));

            // unsafe down casting, we better be sure that the stream really contains a highScoresTable!
            highScoresTable = (HighScoresTable) objectInputStream.readObject();
        } catch (FileNotFoundException e) { // Can't find file to open
            System.err.println("Unable to find file: " + filename);
            return null;
        } catch (ClassNotFoundException e) { // The class in the stream is unknown to the JVM
            System.err.println("Unable to find class for object in file: " + filename);
            return null;
        } catch (IOException e) { // Some other problem
            System.err.println("Failed reading object");
            e.printStackTrace(System.err);
            return null;
        } finally { //closing the stream!
            try {
                if (objectInputStream != null) {
                    objectInputStream.close();
                }
            } catch (IOException e) {
                System.err.println("Failed closing file: " + filename);
            }
        }
        return highScoresTable;
    }

    /**
     * Add a high-score.
     *
     * @param score the score
     */
    public void add(ScoreInfo score) {
        scoresList.add(getRank(score.getScore()) - 1, score);
        if (this.scoresList.size() > this.size) {
            scoresList.remove(size());
        }
    }

    /**
     * Is high score boolean.
     *
     * @param score the score
     * @return the boolean
     */
    public boolean isHighScore(Counter score) {
        if (this.scoresList.size() < this.size()) {
            return true;
        } else { //scoresList size==size of table
            //new score bigger than lowest on list
            if (score.getValue() > this.scoresList.get(this.scoresList.size() - 1).getScore()) {
                return true;
            }
        }
        return false;
    }

    /**
     * Return table size.
     *
     * @return the int
     */
    public int size() {
        return size;
    }

    /**
     * Return the current high scores.
     * The list is sorted such that the highest
     * scores come first.
     *
     * @return the high scores
     */
    public List<ScoreInfo> getHighScores() {
        return this.scoresList;
    }

    /**
     * return the rank of the current score: where will it
     * be on the list if added?
     * Rank 1 means the score will be highest on the list.
     * Rank `size` means the score will be lowest.
     * Rank > `size` means the score is too low and will not
     * be added to the list
     *
     * @param score the score
     * @return the rank
     */
    public int getRank(int score) {
        for (int i = 0; i < this.scoresList.size(); i++) {
            if (score > this.scoresList.get(i).getScore()) {
                return i + 1;
            }
        }
        return this.scoresList.size() + 1;
    }

    /**
     * Clears the table.
     */
    public void clear() {
        this.scoresList.clear();
    }

    /**
     * Load table data from file.
     * Current table data is cleared.
     *
     * @param filename the filename
     * @throws IOException the io exception
     */
    public void load(File filename) throws IOException {
        HighScoresTable highScoresTable = loadFromFile(filename);
        if (highScoresTable == null) {
            throw new IOException("Failed Reading File");
        } else {
            this.scoresList = highScoresTable.scoresList;
        }
    }

    /**
     * Save table data to the specified file.
     *
     * @param filename the filename
     * @throws IOException the io exception
     */
    public void save(File filename) throws IOException {
        ObjectOutputStream objectOutputStream = null;
        try {
            objectOutputStream = new ObjectOutputStream(new FileOutputStream(filename));
            objectOutputStream.writeObject(this);
        } catch (IOException e) {
            System.err.println("Failed saving object");
            e.printStackTrace(System.err);
            throw new IOException(e);
        } finally { //closing the stream!
            try {
                if (objectOutputStream != null) {
                    objectOutputStream.close();
                }
            } catch (IOException e) {
                System.err.println("Failed closing file: " + filename);
            }
        }
    }
}
```

```java
package game;

import animation.Animation;
import animation.AnimationRunner;
import animation.Task;

/**
 * The type showHiScore task.
 *
 * @author Elad Israel
 * @version 4.0 17/06/2018
 */

public class ShowHiScoresTask implements Task<Void> {

    private AnimationRunner runner;
    private Animation highScoresAnimation;


    /**
     * Instantiates a new Show hi scores task.
     *
     * @param runner              the runner
     * @param highScoresAnimation the high scores animation
     */
    public ShowHiScoresTask(AnimationRunner runner, Animation highScoresAnimation) {
        this.runner = runner;
        this.highScoresAnimation = highScoresAnimation;
    }

    /**
     * runs the task.
     *
     * @return T
     */
    public Void run() {
        this.runner.run(this.highScoresAnimation);
        return null;
    }
}
```

```java
package game;

import biuoop.DrawSurface;

import java.util.ArrayList;
import java.util.List;

/**
 * Class name: SpriteCollection
 * a SpriteCollection will hold a collection of sprites.
 *
 * @author Elad Israel
 * @version 3.0 20/05/2018
 */
public class SpriteCollection {
    private List<Sprite> sprites = new ArrayList<>();


    /**
     * add the given sprite to the collection.
     *
     * @param s given sprite.
     */
    public void addSprite(Sprite s) {
        sprites.add(s);
    }

    /**
     * remove the given sprite from the collection.
     *
     * @param s given sprite.
     */
    public void removeSprite(Sprite s) {
        sprites.remove(s);
    }

    /**
     * call timePassed() on all sprites.
     *
     * @param dt amount of seconds passed since the last call
     */
    public void notifyAllTimePassed(double dt) {
        // Make a copy of the Sprites before iterating over them.
        List<Sprite> spritesCopy = new ArrayList<Sprite>(this.sprites);
        // Notify all Sprites that time passed:
        for (Sprite sprite : spritesCopy) {
            sprite.timePassed(dt);
        }
    }

    /**
     * call drawOn(d) on all sprites.
     *
     * @param d drawSurface
     */
    public void drawAllOn(DrawSurface d) {
        for (Sprite sprite : sprites) {
            sprite.drawOn(d);
        }
    }
}
```

```java
package game;

import animation.GameLevel;
import biuoop.DrawSurface;

import java.awt.Color;

/**
 * Class name: NameOfLevelIndicator
 * NameOfLevelIndicator sprite that will sit at the top of the screen and indicate the name of the level.
 *
 * @author Elad Israel
 * @version 3.0 20/05/2018
 */
public class NameOfLevelIndicator implements Sprite {
    private String name;

    /**
     * Constructor.
     *
     * @param name the name
     */
    public NameOfLevelIndicator(String name) {
        this.name = name;
    }

    /**
     * Specify what the ScoreIndicator does when time is passed.
     *
     * @param dt amount of seconds passed since the last call
     */
    public void timePassed(double dt) {

    }

    /**
     * draws the ScoreIndicator on the surface.
     *
     * @param d draw surface
     */
    public void drawOn(DrawSurface d) {
        final int lettersSize = 15;
        d.setColor(Color.black);
        d.drawText(d.getHeight(), 19, "Level Name: " + this.name, lettersSize);
    }

    /**
     * adds the ScoreIndicator to the game-as a sprite.
     *
     * @param g game
     */
    public void addToGame(GameLevel g) {
        g.addSprite(this);
    }
}
```

```java
package levels;

import biuoop.DrawSurface;
import game.Sprite;
import shapes.Rectangle;

/**
 * interface name: Background
 * The Background interface.
 * describes a Background of a level or block.
 *
 * @author Elad Israel
 * @version 4.0 17/06/2018
 */
public interface Background extends Sprite {

    /**
     * Draw Background on drawsurface.
     *
     * @param d the drawsurface
     * @param r the rectangle
     */
    void drawOn(DrawSurface d, Rectangle r);
}
```

```java
package levels;

import game.Block;

/**
 * Creates a block.
 *
 * @author Elad Israel
 * @version 4.0 17/06/2018
 */
public interface BlockCreator {

    /**
     * Create a block at the specified location.
     *
     * @param xpos the x position of the block
     * @param ypos the y position of the block
     * @return block
     */
    Block create(int xpos, int ypos);
}
```

```java
package levels;

import java.awt.Color;
import java.util.HashMap;
import java.util.Map;

/**
 * parse color definition and return the specified color.
 * parse from string of the structure: "colorname" or "RGB(x,y,z)"
 * @author Elad Israel
 * @version 4.0 17/06/2018
 */
public class ColorsParser {

    /**
     * parse color definition and return the specified color.
     * parse from string of the structure: "colorname" or "RGB(x,y,z)"
     *
     * @param s the s
     * @return Color color
     */
    public static java.awt.Color colorFromString(String s) {
        if (s.startsWith("RGB")) {
            int x, y, z;
            s = s.substring("RGB".length()).trim();
            s = s.substring(1, s.length() - 1); // removes "(" ")"
            String[] rgbVals = s.split(",");
            try {
                x = Integer.parseInt(rgbVals[0]);
                y = Integer.parseInt(rgbVals[1]);
                z = Integer.parseInt(rgbVals[2]);
            } catch (Exception e) {
                throw new RuntimeException("invalid Color RGB");
            }
            return new Color(x, y, z);
        } else {
            Map<String, Color> colorsMap = new HashMap<String, Color>();
            colorsMap.put("yellow", Color.yellow);
            colorsMap.put("red", Color.red);
            colorsMap.put("black", Color.black);
            colorsMap.put("blue", Color.blue);
            colorsMap.put("cyan", Color.cyan);
            colorsMap.put("gray", Color.gray);
            colorsMap.put("lightGray", Color.lightGray);
            colorsMap.put("green", Color.green);
            colorsMap.put("orange", Color.orange);
            colorsMap.put("pink", Color.pink);
            colorsMap.put("white", Color.white);
            if (colorsMap.get(s) == null) {
                throw new RuntimeException("invalid Color name");
            }
            return colorsMap.get(s);
        }
    }
}
```

```java
package levels;

import game.Block;
import game.Velocity;

import java.util.List;


/**
 * produces a level.
 *
 * @author Elad Israel
 * @version 4.0 17/06/2018
 */
public class LevelFactory implements LevelInformation {
    private int numberOfBalls;
    private List<Velocity> initialBallVelocities;
    private int paddleSpeed;
    private int paddleWidth;
    private String levelName;
    private Background getBackground;
    private List<Block> blocks;
    private int numberOfBlocksToRemove;

    /**
     * Instantiates a new Level factory.
     */
    public LevelFactory() {
        this.numberOfBalls = -1;
        this.initialBallVelocities = null;
        this.paddleSpeed = -1;
        this.paddleWidth = -1;
        this.levelName = null;
        this.getBackground = null;
        this.blocks = null;
        this.numberOfBlocksToRemove = -1;
    }

    /**
     * Is all set boolean.
     *
     * @return the boolean
     */
    public Boolean isAllSet() {
        if (this.numberOfBalls != -1
                && this.initialBallVelocities != null
                && this.paddleSpeed != -1
                && this.paddleWidth != -1
                && this.levelName != null
                && this.getBackground != null
                && this.blocks != null
                && this.numberOfBlocksToRemove != -1) {
            return true;
        }
        return false;
    }

    /**
     * Sets number of balls.
     *
     * @param numberOfBallsToSet the number of balls
     */
    public void setNumberOfBalls(int numberOfBallsToSet) {
        this.numberOfBalls = numberOfBallsToSet;
    }

    /**
     * Sets initial ball velocities.
     *
     * @param initialBallVelocitiesToSet the initial ball velocities
     */
    public void setInitialBallVelocities(List<Velocity> initialBallVelocitiesToSet) {
        this.initialBallVelocities = initialBallVelocitiesToSet;
        setNumberOfBalls(initialBallVelocities.size());
    }

    /**
     * Sets paddle speed.
     *
     * @param paddleSpeedToSet the paddle speed
     */
    public void setPaddleSpeed(int paddleSpeedToSet) {
        this.paddleSpeed = paddleSpeedToSet;
    }

    /**
     * Sets paddle width.
     *
     * @param paddleWidthToSet the paddle width
     */
    public void setPaddleWidth(int paddleWidthToSet) {
        this.paddleWidth = paddleWidthToSet;
    }

    /**
     * Sets level name.
     *
     * @param levelNameToSet the level name
     */
    public void setLevelName(String levelNameToSet) {
        this.levelName = levelNameToSet;
    }

    /**
     * Sets blocks.
     *
     * @param blocksToSet the blocks
     */
    public void setBlocks(List<Block> blocksToSet) {
        this.blocks = blocksToSet;
    }

    /**
     * Sets number of blocks to remove.
     *
     * @param numOfBlocksToRemove the number of blocks to remove
     */
    public void setNumberOfBlocksToRemove(int numOfBlocksToRemove) {
        this.numberOfBlocksToRemove = numOfBlocksToRemove;
    }

    /**
     * Number of balls int.
     *
     * @return the int
     */
    public int numberOfBalls() {
        return numberOfBalls;
    }

    /**
     * The initial velocity of each ball.
     * Note that initialBallVelocities().size() == numberOfBalls()
     *
     * @return the list of velocities
     */
    public List<Velocity> initialBallVelocities() {
        return initialBallVelocities;
    }

    /**
     * Paddle speed int.
     *
     * @return the int
     */
    public int paddleSpeed() {
        return paddleSpeed;
    }

    /**
     * Paddle width int.
     *
     * @return the int
     */
    public int paddleWidth() {
        return paddleWidth;
    }

    /**
     * Level name string.
     * the level name will be displayed at the top of the screen.
     *
     * @return the string
     */
    public String levelName() {
        return levelName;
    }

    /**
     * Returns a sprite with the background of the level.
     *
     * @return the background
     */
    public Background getBackground() {
        return getBackground;
    }

    /**
     * Sets background.
     *
     * @param backgroundToSet the get background
     */
    public void setBackground(Background backgroundToSet) {
        this.getBackground = backgroundToSet;
    }

    /**
     * The Blocks that make up this level, each block contains its size, color and location.
     *
     * @return Blocks list
     */
    public List<Block> blocks() {
        return blocks;
    }

    /**
     * Number of levels that should be removed before the level is considered to be "cleared".
     * This number should be <= blocks.size();
     *
     * @return the int
     */
    public int numberOfBlocksToRemove() {
        return numberOfBlocksToRemove;
    }
}
```

```java
package levels;

import biuoop.DrawSurface;
import shapes.Rectangle;

import java.awt.Color;

/**
 * Background with Color.
 * describes a Background of a level or block with color.
 *
 * @author Elad Israel
 * @version 4.0 17/06/2018
 */
public class BackgroundColor implements Background {
    private Color color;

    /**
     * Instantiates a new Background color.
     *
     * @param color the color
     */
    public BackgroundColor(Color color) {
        this.color = color;
    }

    /**
     * draw the sprite to the screen.
     *
     * @param d drawSurface
     */
    public void drawOn(DrawSurface d) {
        drawOn(d, null);
    }

    /**
     * Draw Background on drawsurface.
     *
     * @param d the drawsurface
     * @param rectangle the rectangle
     */
    public void drawOn(DrawSurface d, Rectangle rectangle) {
        d.setColor(color);
        if (rectangle == null) {
            d.fillRectangle(0, 0, d.getWidth(), d.getHeight());
        } else {
            d.fillRectangle((int) rectangle.getUpperLeft().getX(), (int) rectangle.getUpperLeft().getY(),
                    (int) rectangle.getWidth(), (int) rectangle.getHeight());
        }
    }


    /**
     * notify the sprite that time has passed.
     * @param dt amount of seconds passed since the last call
     */
    public void timePassed(double dt) {

    }
}
```

```java
package levels;


import biuoop.DrawSurface;
import shapes.Rectangle;

import javax.imageio.ImageIO;
import java.awt.Image;
import java.io.IOException;

/**
 * Background with Image.
 * describes a Background of a level or block with image.
 *
 * @author Elad Israel
 * @version 4.0 17/06/2018
 */
public class BackgroundImage implements Background {
    private Image image;

    /**
     * Instantiates a new Background image.
     *
     * @param path the path
     */
    public BackgroundImage(String path) {
        try {
            image = ImageIO.read(ClassLoader.getSystemClassLoader().getResourceAsStream(path));
        } catch (IOException e) {
            throw new RuntimeException("Cannot read image file!");
        }
    }

    /**
     * draw the sprite to the screen.
     *
     * @param d drawSurface
     */
    public void drawOn(DrawSurface d) {
        drawOn(d, null);
    }

    /**
     * Draw Background on drawsurface.
     *
     * @param d         the drawsurface
     * @param rectangle the rectangle
     */
    public void drawOn(DrawSurface d, Rectangle rectangle) {
        if (rectangle == null) {
            d.drawImage(0, 0, image);
        } else {
            d.drawImage((int) rectangle.getUpperLeft().getX(), (int) rectangle.getUpperLeft().getY(), image);
        }
    }


    /**
     * notify the sprite that time has passed.
     *
     * @param dt amount of seconds passed since the last call
     */
    public void timePassed(double dt) {

    }
}
```

```java
package levels;

import game.Block;
import game.Sprite;
import game.Velocity;

import java.util.List;

/**
 * interface name: LevelInformation
 * The LevelInformation interface specifies the information required to fully describe a level.
 *
 * @author Elad Israel
 * @version 3.0 20/05/2018
 */
public interface LevelInformation {
    /**
     * Number of balls int.
     *
     * @return the int
     */
    int numberOfBalls();

    /**
     * The initial velocity of each ball.
     * Note that initialBallVelocities().size() == numberOfBalls()
     *
     * @return the list of velocities
     */
    List<Velocity> initialBallVelocities();

    /**
     * Paddle speed int.
     *
     * @return the int
     */
    int paddleSpeed();

    /**
     * Paddle width int.
     *
     * @return the int
     */
    int paddleWidth();

    /**
     * Level name string.
     * the level name will be displayed at the top of the screen.
     *
     * @return the string
     */
    String levelName();

    /**
     * Returns a sprite with the background of the level.
     *
     * @return the background
     */
    Sprite getBackground();

    /**
     * The Blocks that make up this level, each block contains its size, color and location.
     *
     * @return Blocks list
     */
    List<Block> blocks();

    /**
     * Number of levels that should be removed before the level is considered to be "cleared".
     * This number should be <= blocks.size();
     *
     * @return the int
     */
    int numberOfBlocksToRemove();
}
```

```java
package levels;


import game.Block;
import shapes.Point;
import shapes.Rectangle;

import java.awt.Color;
import java.util.HashMap;
import java.util.Map;

/**
 * each instance of this class generates a different type of block from the block definitions.
 *
 * @author Elad Israel
 * @version 4.0 17/06/2018
 */
public class BlockTemplateCreator implements BlockCreator {
    private int width;
    private int height;
    private int hitPoints;
    private Map<Integer, Background> hpToBackground;
    private Color stroke;
    private Background background;

    /**
     * Instantiates a new Block template creator.
     */
    public BlockTemplateCreator() {
        this.width = -1;
        this.height = -1;
        this.hitPoints = -1;
        this.hpToBackground = new HashMap<>();
        this.background = null;
        this.stroke = null;
    }

    /**
     * Sets background.
     *
     * @param backgroundToSet the background
     */
    public void setBackground(Background backgroundToSet) {
        this.background = backgroundToSet;
    }

    /**
     * Sets width.
     *
     * @param widthToSet the width
     */
    public void setWidth(int widthToSet) {
        this.width = widthToSet;
    }

    /**
     * Sets height.
     *
     * @param heightToSet the height
     */
    public void setHeight(int heightToSet) {
        this.height = heightToSet;
    }

    /**
     * Sets hit points.
     *
     * @param hitPointsToSet the hit points
     */
    public void setHitPoints(int hitPointsToSet) {
        this.hitPoints = hitPointsToSet;
    }

    /**
     * Add hp to background.
     *
     * @param hp          the hp
     * @param backgroundToSet the background
     */
    public void addHpToBackground(Integer hp, Background backgroundToSet) {
        this.hpToBackground.put(hp, backgroundToSet);
    }

    /**
     * Sets stroke.
     *
     * @param strokeToSet the stroke
     */
    public void setStroke(Color strokeToSet) {
        this.stroke = strokeToSet;
    }

    /**
     * Create a block at the specified location.
     *
     * @param xpos the x position of the block
     * @param ypos the y position of the block
     * @return block
     */
    public Block create(int xpos, int ypos) {
        Block block = new Block(new Rectangle(new Point(xpos, ypos), this.width, this.height));
        if (width < 0 || height < 0 || hitPoints < 0) {
            throw new RuntimeException("block wasn't initialized properly");
        }
        block.setHitPoints(this.hitPoints);
        block.setBackground(this.background);
        block.setHpToBackground(this.hpToBackground);
        if (this.stroke != null) {
            block.setStroke(this.stroke);
        }
        for (int hitPoint : this.hpToBackground.keySet()) {
            block.addHpBackground(hitPoint, this.hpToBackground.get(hitPoint));
        }
        return block;
    }
}
```

```java
package levels;

import java.io.BufferedReader;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

/**
 * in charge of reading a block-definitions file and returning a BlocksFromSymbolsFactory object.
 *
 * @author Elad Israel
 * @version 4.0 17/06/2018
 */
public class BlocksDefinitionReader {

    private Map<String, Integer> spacerWidths;
    private Map<String, BlockCreator> blockCreators;


    /**
     * Instantiates a new Blocks definition reader.
     */
    public BlocksDefinitionReader() {
        this.spacerWidths = new HashMap<>();
        this.blockCreators = new HashMap<>();
    }


    /**
     * get a symbol and create the desired block.
     *
     * @param reader the reader
     * @return the blocks from symbols factory
     */
    public static BlocksFromSymbolsFactory fromReader(java.io.Reader reader) {
        BufferedReader bufferReader = new BufferedReader(reader);
        Map<String, String> defaultValues = new HashMap<>();
        BlocksDefinitionReader bdr = new BlocksDefinitionReader();
        String line;
        try {
            line = bufferReader.readLine();
        } catch (Exception e) {
            try {
                bufferReader.close();
            } catch (IOException e1) {
                e1.printStackTrace();
            }
            throw new RuntimeException("reading Blocks definition failed!");
        }
        while (line != null) {
            if (line.equals("") || line.startsWith("#")) {
                try {
                    line = bufferReader.readLine();
                    continue;
                } catch (Exception e) {
                    try {
                        bufferReader.close();
                    } catch (IOException e1) {
                        e1.printStackTrace();
                    }
                    throw new RuntimeException("reading Blocks definition failed!");
                }
            }
            if (line.startsWith("default")) {
                line = line.substring("default".length()).trim();
                defaultValues.putAll(stringLineToMap(line));
            } else if (line.startsWith("bdef")) { //block definitions
                line = line.substring("bdef".length()).trim();
                Map<String, String> blockDefMap = stringLineToMap(line);
                String symbol = blockDefMap.get("symbol");
                if (symbol.length() != 1) {
                    try {
                        bufferReader.close();
                    } catch (IOException e1) {
                        e1.printStackTrace();
                    }
                    throw new RuntimeException("invalid symbol");
                }
                blockDefMap.putAll(defaultValues);
                BlockTemplateCreator blockTemplate = new BlockTemplateCreator();
                for (String key : blockDefMap.keySet()) {
                    if (key.equals("width")) {
                        try {
                            blockTemplate.setWidth(Integer.parseInt(blockDefMap.get(key)));
                        } catch (Exception e) {
                            try {
                                bufferReader.close();
                            } catch (IOException e1) {
                                e1.printStackTrace();
                            }
                            throw new RuntimeException("parsing to int failed");
                        }
                    } else if (key.equals("height")) {
                        try {
                            blockTemplate.setHeight(Integer.parseInt(blockDefMap.get(key)));
                        } catch (Exception e) {
                            throw new RuntimeException("parsing to int failed");
                        }
                    } else if (key.equals("hit_points")) {
                        try {
                            blockTemplate.setHitPoints(Integer.parseInt(blockDefMap.get(key)));
                        } catch (Exception e) {
                            try {
                                bufferReader.close();
                            } catch (IOException e1) {
                                e1.printStackTrace();
                            }
                            throw new RuntimeException("parsing to int failed");
                        }
                    } else if (key.startsWith("fill")) {
                        String fillKey = key.substring("fill".length());
                        String value = blockDefMap.get(key);
                        Background background;
                        if (value.startsWith("color")) {
                            value = value.substring("color".length()).trim();
                            value = value.substring(1, value.length() - 1); // removes "(" ")"
                            background = new BackgroundColor(ColorsParser.colorFromString(value));
                        } else if (value.startsWith("image")) {
                            value = value.substring("image".length()).trim();
                            value = value.substring(1, value.length() - 1); // removes "(" ")"
                            background = new BackgroundImage(value);
                        } else {
                            throw new RuntimeException("invalid fill parameter");
                        }
                        if (fillKey.startsWith("-")) {
                            int fillHP = Integer.parseInt(fillKey.substring(1));
                            blockTemplate.addHpToBackground(fillHP, background);
                        } else {
                            blockTemplate.setBackground(background);
                        }
                    } else if (key.equals("stroke")) {
                        String stroke = blockDefMap.get(key);
                        stroke = stroke.substring("color".length()).trim();
                        stroke = stroke.substring(1, stroke.length() - 1); // removes "(" ")"
                        blockTemplate.setStroke(ColorsParser.colorFromString(stroke));
                    }
                }
                //adds a block type(symbol and block template to create from that symbol) to BlockCreators list.
                bdr.blockCreators.put(symbol, blockTemplate);
            } else if (line.startsWith("sdef symbol:")) { //spacer definitions
                line = line.substring("sdef symbol:".length()).trim();
                String symbol = line.substring(0, 1);
                line = line.substring(1).trim();
                if (!line.startsWith("width:")) {
                    try {
                        bufferReader.close();
                    } catch (IOException e1) {
                        e1.printStackTrace();
                    }
                    throw new RuntimeException("invalid spacer parameters");
                }
                int width;
                try {
                    width = Integer.parseInt(line.substring("width:".length()));
                } catch (Exception e) {
                    try {
                        bufferReader.close();
                    } catch (IOException e1) {
                        e1.printStackTrace();
                    }
                    throw new RuntimeException("spacer parsing to int failed");
                }
                bdr.spacerWidths.put(symbol, width);
            } else {
                try {
                    bufferReader.close();
                } catch (IOException e1) {
                    e1.printStackTrace();
                }
                throw new RuntimeException("unknown characters in blocks definition");
            }
            try {
                line = bufferReader.readLine();
            } catch (Exception e) {
                try {
                    bufferReader.close();
                } catch (IOException e1) {
                    e1.printStackTrace();
                }
                throw new RuntimeException("reading Blocks definition failed!");
            }
        }
        return new BlocksFromSymbolsFactory(bdr.spacerWidths, bdr.blockCreators);
    }


    /**
     * String line to map.
     *
     * @param line the line
     * @return the map
     */
    public static Map<String, String> stringLineToMap(String line) {
        Map<String, String> map = new HashMap<>();
        String[] pairs = line.split(" ");
        for (String pair : pairs) {
            String[] keyValuePair = pair.split(":");
            if (keyValuePair.length != 2) {
                throw new RuntimeException("converting string to map failed!");
            }
            map.put(keyValuePair[0], keyValuePair[1]);
        }
        return map;
    }

    /**
     * Get spacer widths map.
     *
     * @return the map
     */
    public Map<String, Integer> getSpacerWidths() {
        return this.spacerWidths;
    }

    /**
     * Get block creators map.
     *
     * @return the map
     */
    public Map<String, BlockCreator> getBlockCreators() {
        return this.blockCreators;
    }
}
```

```java
package levels;

import game.Block;

import java.util.Map;

/**
 * Creates Blocks from symbols.
 * (has a method which gets a symbol and create the desired block).
 * The block definition files define a mapping from symbols to spaces and blocks. These symbols are then used in the
 * level specification files to define the blocks that need to be created.
 * @author Elad Israel
 * @version 4.0 17/06/2018
 */
public class BlocksFromSymbolsFactory {
    private Map<String, Integer> spacerWidths;
    private Map<String, BlockCreator> blockCreators;

    /**
     * Instantiates a new Blocks from symbols factory.
     *
     * @param spacerWidths  the spacer widths
     * @param blockCreators the block creators
     */
    public BlocksFromSymbolsFactory(Map<String, Integer> spacerWidths, Map<String, BlockCreator> blockCreators) {
        this.spacerWidths = spacerWidths;
        this.blockCreators = blockCreators;
    }


    /**
     * returns true if 's' is a valid space symbol.
     *
     * @param s the s
     * @return the boolean
     */
    public boolean isSpaceSymbol(String s) {
        return spacerWidths.containsKey(s);
    }

    /**
     * returns true if 's' is a valid block symbol.
     *
     * @param s the s
     * @return the boolean
     */
    public boolean isBlockSymbol(String s) {
        return blockCreators.containsKey(s);
    }

    /**
     * Returns the width in pixels associated with the given spacer-symbol.
     *
     * @param s the s
     * @return the int
     */
    public int getSpaceWidth(String s) {
        return this.spacerWidths.get(s);
    }

    /**
     * Return a block according to the definitions associated
     * with symbol s. The block will be located at position (x, y).
     *
     * @param symbol the symbol
     * @param x      the x
     * @param y      the y
     * @return the block
     */
    public Block getBlock(String symbol, int x, int y) {
        return this.blockCreators.get(symbol).create(x, y);
    }
}
```

```java
package levels;


import game.Block;
import game.Velocity;

import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.Reader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;


/**
 * The type Level specification reader.
 *
 * @author Elad Israel
 * @version 4.0 17/06/2018
 */
public class LevelSpecificationReader {


    /**
     * get a file name and returns a list of LevelInformation objects.
     *
     * @param reader to the file
     * @return a list of LevelInformation objects
     */
    public List<LevelInformation> fromReader(java.io.Reader reader) {
        List<String> lines = new ArrayList<>();
        BufferedReader bufferReader = new BufferedReader(reader);
        try {
            String line = bufferReader.readLine();
            while (line != null) {
                if (line.equals("") || line.startsWith("#")) {
                    line = bufferReader.readLine();
                    continue;
                }
                lines.add(line);
                line = bufferReader.readLine();
            }
        } catch (Exception e) {
            try {
                bufferReader.close();
            } catch (IOException e1) {
                e1.printStackTrace();
            }
            throw new RuntimeException("reading Blocks definition failed!");
        }
        List<List<String>> levels = separateLevels(lines);
        List<LevelInformation> levelInformationList = new ArrayList<>();
        for (List<String> level : levels) {
            levelInformationList.add(levelToLevelInformation(level));
        }
        return levelInformationList;
    }

    /**
     * receives all lines in a LevelSpecification file and separates them into list containing lists-
     * each internal list contains all the information of a specific level.
     *
     * @param lines the lines
     * @return list
     */
    public List<List<String>> separateLevels(List<String> lines) {
        List<List<String>> levels = new ArrayList<>();
        Boolean insideLevel = false;
        for (int i = 0, levelNum = 0; i < lines.size(); i++) {
            if (lines.get(i).equals("START_LEVEL")) {
                if (insideLevel) {
                    throw new RuntimeException("START_LEVEL twice");
                }
                insideLevel = true;
                //add a new level
                levels.add(new ArrayList<>());
            } else if (lines.get(i).equals("END_LEVEL")) {
                if (!insideLevel) {
                    throw new RuntimeException("END_LEVEL twice");
                }
                insideLevel = false;
                levelNum++;
            } else if (insideLevel) {
                levels.get(levelNum).add(lines.get(i));
            }
        }
        return levels;
    }

    /**
     * Understanding the content of the level specification of a single level: this will go over the strings,
     * split and parse them, and map them to java objects, resulting in a LevelInformation object.
     *
     * @param stringsOfLevel the strings of level
     * @return LevelInformation object.
     */
    public LevelInformation levelToLevelInformation(List<String> stringsOfLevel) {
        LevelFactory level = new LevelFactory();

        //extract blocks rows
        List<String> blocksLayoutInLevel = extractBlocksLines(stringsOfLevel);
        //extract level properties
        extractAndSetLevelProperties(stringsOfLevel, level);
        //extract block properties: row_height, block_start_x, block_start_y.
        Map<String, Integer> blocksProperties = extractBlocksProperties(stringsOfLevel);
        //extract Block Definitions file
        String blocksDefFilePath = null;
        for (String line : stringsOfLevel) {
            if (line.startsWith("block_definitions:")) {
                line = line.substring("block_definitions:".length()).trim();
                blocksDefFilePath = line;
            }
        }
        if (blocksDefFilePath == null) {
            throw new RuntimeException("block_definitions path wasn't found in level spec");
        }
        BlocksFromSymbolsFactory blocksFromSymbolsFactory = blocksDefToFactory(blocksDefFilePath);
        List<Block> blocks = getBlocks(blocksFromSymbolsFactory, blocksLayoutInLevel, blocksProperties);
        level.setBlocks(blocks);
        if (!level.isAllSet()) {
            throw new RuntimeException("level property missing");
        }
        return level;
    }

    /**
     * Extract blocks lines list.
     *
     * @param stringsOfLevel the strings of level
     * @return the list
     */
    public List<String> extractBlocksLines(List<String> stringsOfLevel) {
        List<String> blockLines = new ArrayList<>();
        Boolean insideBlocks = false;
        for (String line : stringsOfLevel) {
            if (line.equals("START_BLOCKS")) {
                if (insideBlocks) {
                    throw new RuntimeException("START_BLOCKS twice");
                }
                insideBlocks = true;
            } else if (line.equals("END_BLOCKS")) {
                if (!insideBlocks) {
                    throw new RuntimeException("END_BLOCKS twice");
                }
                insideBlocks = false;
            } else if (insideBlocks) {
                //add a new line of block
                blockLines.add(line);
            }
        }
        return blockLines;
    }


    /**
     * Extract and set level properties.
     *
     * @param stringsOfLevel the strings of level
     * @param level          the level
     */
    public void extractAndSetLevelProperties(List<String> stringsOfLevel, LevelFactory level) {
        for (String line : stringsOfLevel) {
            if (line.startsWith("level_name:")) {
                line = line.substring("level_name:".length()).trim();
                level.setLevelName(line);
            } else if (line.startsWith("ball_velocities:")) {
                line = line.substring("ball_velocities:".length()).trim();
                Map<String, String> angleSpeedPairs = stringLineToMap(line);
                List<Velocity> ballVelocities = new ArrayList<>();
                for (String angle : angleSpeedPairs.keySet()) {
                    try {
                        ballVelocities.add(new Velocity(Velocity.fromAngleAndSpeed(Integer.parseInt(angle),
                                Integer.parseInt(angleSpeedPairs.get(angle)))));
                    } catch (Exception e) {
                        throw new RuntimeException("invalid ball_velocities");
                    }
                }
                level.setInitialBallVelocities(ballVelocities);

            } else if (line.startsWith("background:")) {
                line = line.substring("background:".length()).trim();
                if (line.startsWith("image")) {
                    line = line.substring("image".length()).trim();
                    line = line.substring(1, line.length() - 1); // removes "(" ")"
                    level.setBackground(new BackgroundImage(line));
                } else if (line.startsWith("color")) {
                    line = line.substring("color".length()).trim();
                    line = line.substring(1, line.length() - 1); // removes "(" ")"
                    level.setBackground(new BackgroundColor(ColorsParser.colorFromString(line)));
                } else {
                    throw new RuntimeException("invalid background parameters");
                }
            } else if (line.startsWith("paddle_speed:")) {
                line = line.substring("paddle_speed:".length()).trim();
                try {
                    level.setPaddleSpeed(Integer.parseInt(line));
                    if (level.paddleSpeed() < 0) {
                        throw new Exception();
                    }
                } catch (Exception e) {
                    throw new RuntimeException("invalid paddle_speed");
                }
            } else if (line.startsWith("paddle_width:")) {
                line = line.substring("paddle_width:".length()).trim();
                try {
                    level.setPaddleWidth(Integer.parseInt(line));
                    if (level.paddleWidth() < 0) {
                        throw new Exception();
                    }
                } catch (Exception e) {
                    throw new RuntimeException("invalid paddle_width");
                }
            } else if (line.startsWith("num_blocks:")) {
                line = line.substring("num_blocks:".length()).trim();
                try {
                    level.setNumberOfBlocksToRemove(Integer.parseInt(line));
                    if (level.numberOfBlocksToRemove() < 0) {
                        throw new Exception();
                    }
                } catch (Exception e) {
                    throw new RuntimeException("invalid num_blocks");
                }
            }
        }
    }

    /**
     * String line to map map.
     *
     * @param line the line
     * @return the map
     */
    public Map<String, String> stringLineToMap(String line) {
        Map<String, String> map = new HashMap<>();
        String[] pairs = line.split(" ");
        for (String pair : pairs) {
            String[] keyValuePair = pair.split(",");
            if (keyValuePair.length != 2) {
                throw new RuntimeException("converting string to map failed!");
            }
            map.put(keyValuePair[0], keyValuePair[1]);
        }
        return map;
    }


    /**
     * Extract blocks properties map.
     *
     * @param stringsOfLevel the strings of level
     * @return the map
     */
    public Map<String, Integer> extractBlocksProperties(List<String> stringsOfLevel) {
        Map<String, Integer> blocksProperties = new HashMap<>();
        for (String line : stringsOfLevel) {
            if (line.startsWith("blocks_start_x:")) {
                line = line.substring("blocks_start_x:".length()).trim();
                try {
                    blocksProperties.put("blocks_start_x", Integer.parseInt(line));
                } catch (Exception e) {
                    throw new RuntimeException("invalid blocks properties");
                }
            } else if (line.startsWith("blocks_start_y:")) {
                line = line.substring("blocks_start_y:".length()).trim();
                try {
                    blocksProperties.put("blocks_start_y", Integer.parseInt(line));
                } catch (Exception e) {
```

```java
                    throw new RuntimeException("invalid blocks properties");
                }
            } else if (line.startsWith("row_height:")) {
                line = line.substring("row_height:".length()).trim();
                try {
                    blocksProperties.put("row_height", Integer.parseInt(line));
                } catch (Exception e) {
                    throw new RuntimeException("invalid blocks properties");
                }
            }
        }
        if (blocksProperties.size() != 3) {
            throw new RuntimeException("missing blocks properties");
        }
        return blocksProperties;
    }


    /**
     * Blocks definitions to BlocksFromSymbolsFactory.
     *
     * @param blocksDefFilePath the blocks def file path
     * @return the blocks from symbols factory
     */
    public BlocksFromSymbolsFactory blocksDefToFactory(String blocksDefFilePath) {
        Reader reader;
        InputStream is = null;
        try {

            is = ClassLoader.getSystemClassLoader().getResourceAsStream(blocksDefFilePath);
            // Reading the blocks.
            reader = new BufferedReader(
                    new InputStreamReader(is));
        } catch (Exception e) {
            if (is != null) {
                try {
                    is.close();
                } catch (IOException e1) {
                    e1.printStackTrace();
                }
            }
            throw new RuntimeException("couldn't read blocks definitions file");
        }
        return BlocksDefinitionReader.fromReader(reader);

    }


    /**
     * Gets blocks.
     *
     * @param blocksFromSymbolsFactory the blocks from symbols factory
     * @param blocksLayoutInLevel      the blocks layout in level
     * @param blocksProperties         the blocks properties
     * @return the blocks
     */
    public List<Block> getBlocks(BlocksFromSymbolsFactory blocksFromSymbolsFactory, List<String> blocksLayoutInLevel,
                                 Map<String, Integer> blocksProperties) {
        List<Block> blocks = new ArrayList<>();
        //starting y of first row
        int yPos = blocksProperties.get("blocks_start_y");
        String symbol; //represent block or spacer
        for (String blocksRow : blocksLayoutInLevel) {
            //we cut the row with each iteration so its size is changing. so we need to keep the size beforehand.
            int rowLength = blocksRow.length();
            int xpos = blocksProperties.get("blocks_start_x");
            for (int i = 0; i < rowLength; i++) {
                //extract first symbol
                symbol = blocksRow.substring(0, 1);
                if (blocksFromSymbolsFactory.isBlockSymbol(symbol)) {
                    Block block = blocksFromSymbolsFactory.getBlock(symbol, xpos, yPos);
                    blocks.add(block);
                    xpos += block.getCollisionRectangle().getWidth();
                } else if (blocksFromSymbolsFactory.isSpaceSymbol(symbol)) {
                    xpos += blocksFromSymbolsFactory.getSpaceWidth(symbol);
                } else {
                    throw new RuntimeException("unknown symbol");
                }
                //remove first char
                blocksRow = blocksRow.substring(1);
            }
            yPos += blocksProperties.get("row_height");
        }
        return blocks;
    }
}
```

```java
package shapes;

import biuoop.DrawSurface;
import game.CollisionInfo;
import game.GameEnvironment;
import animation.GameLevel;
import game.Velocity;
import game.Paddle;
import game.Sprite;

import java.awt.Color;

/**
 * Classname: Ball
 * a Ball (actually, a circle) has size (radius), color, and location (a Point).
 * Balls also know how to draw themselves on a DrawSurface.
 *
 * @author Elad Israel
 * @version 3.0 20/05/2018
 */
public class Ball implements Sprite {
    //members
    private int size;
    private Point point;
    private java.awt.Color fillColor;
    private java.awt.Color drawColor;
    private Velocity velocity;
    private GameEnvironment gameEnvironment;

    /**
     * Constructor 1.
     * Constructs a Ball using center point, radius, and color.
     *
     * @param center    center point of this ball.
     * @param r         radius of this ball.
     * @param fillColor fill color of the ball.
     * @param drawColor draw color of the ball.
     */
    public Ball(Point center, int r, java.awt.Color fillColor, java.awt.Color drawColor) {
        this.size = r;
        this.point = center;
        this.fillColor = fillColor;
        this.drawColor = drawColor;
    }

    /**
     * Constructor 2.
     * Constructs a Ball using x and y coordinates of the center point, radius, and color.
     *
     * @param x         x coordinate of the center point of this ball.
     * @param y         y coordinate of the center point of this ball.
     * @param r         radius of this ball.
     * @param fillColor color of this ball's filling.
     * @param drawColor color of this ball's circumference.
     */
    public Ball(int x, int y, int r, java.awt.Color fillColor, java.awt.Color drawColor) {
        this.size = r;
        this.point = new Point(x, y);
        this.fillColor = fillColor;
        this.drawColor = drawColor;
    }

    /**
     * Access method- Return the x value of this ball.
     *
     * @return x coordinate of the center point of this ball.
     */
    public int getX() {
        return (int) this.point.getX();
    }

    /**
     * Access method- Return the y value of this ball.
     *
     * @return y coordinate of the center point of this ball.
     */
    public int getY() {
        return (int) this.point.getY();
    }

    /**
     * Access method- Return the size(radius) of this ball.
     *
     * @return the size(radius) of this ball.
     */
    public int getSize() {
        return this.size;
    }

    /**
```

```java
     * Access method- Return the ball's fill Color.
     *
     * @return the ball's fill Color.
     */
    public java.awt.Color getFillColor() {
        return this.fillColor;
    }

    /**
     * Access method- Return the ball's draw Color.
     *
     * @return the ball's draw Color.
     */
    public java.awt.Color getDrawColor() {
        return this.drawColor;
    }

    /**
     * setter for gameEnviroment.
     *
     * @param gameEnvironmentToSet gameEnvironment to set
     */
    public void setGameEnvironment(GameEnvironment gameEnvironmentToSet) {
        this.gameEnvironment = gameEnvironmentToSet;
    }

    /**
     * draws this ball on the given DrawSurface.
     *
     * @param surface drawSurface
     */
    public void drawOn(DrawSurface surface) {
        //default color if no color was entered
        if (this.fillColor == null || this.drawColor == null) {
            this.fillColor = Color.black;
            this.drawColor = Color.black;
        }
        surface.setColor(this.fillColor);
        surface.fillCircle((int) this.point.getX(), (int) this.point.getY(), this.size);
        surface.setColor(this.drawColor);
        surface.drawCircle((int) this.point.getX(), (int) this.point.getY(), this.size);

    }

    /**
     * Specify what the ball does when time is passed - moves one step.
     *
     * @param dt amount of seconds passed since the last call.
     */
    public void timePassed(double dt) {
        this.moveOneStep(dt);
    }

    /**
     * sets the Velocity of the ball using dx and dy.
     *
     * @param dx dx value to set to this ball's velocity
     * @param dy dy value to set to this ball's velocity
     */
    public void setVelocity(double dx, double dy) {
        this.velocity = new Velocity(dx, dy);
    }

    /**
     * gets the Velocity of the ball.
     *
     * @return velocity
     */
    public Velocity getVelocity() {
        return this.velocity;
    }

    /**
     * sets the Velocity of the ball using Velocity.
     *
     * @param v Velocity value to set to this ball's velocity
     */
    public void setVelocity(Velocity v) {
        this.velocity = v;
    }

    /**
     * calculates the ball speed according to its size(bigger==slower).
     *
     * @param ballSize the ball's size
     * @return speed
     */
    public int speedAccToSize(int ballSize) {
        if (ballSize > 20) {
            return 50;
        }
        if (ballSize > 10) {
            return 150;
        }
        if (ballSize > 7) {
            return 250;
        }
        if (ballSize > 4) {
            return 350;
        }
        if (ballSize > 2) {
            return 450;
        }
        return 550;
    }


    /**
     * calculates where the ball should advance to next:
     * makes sure the ball does not go outside of the screen
     * when it hits the border to the left or to the right, it changes its horizontal direction,
     * and when it hits the border on the top or the bottom, it changes its vertical direction.
     * and then calls applyToPoint that actually moves the ball.
     *
     * @param dt amount of seconds passed since the last call
     */
    public void moveOneStep(double dt) {
        Line trajectory = calculateTrajectory(dt);
        //gets closest collision point to the start of the trajectory line
        CollisionInfo collision;
        try {
            collision = gameEnvironment.getClosestCollision(trajectory);
        } catch (RuntimeException nullPointer) {
            throw new RuntimeException("Ball's gameEnvironment wasn't initialized!");
        }
        //no collision occurred- move the ball regularly to the end of the trajectory
        if (collision == null) {
            this.point = this.velocity.applyToPoint(this.point, dt);
        } else { //collision happened(about to)
            Velocity previous = new Velocity(this.velocity);
            this.velocity = new Velocity(
                    collision.collisionObject().hit(this, collision.collisionPoint(), this.velocity));
            //second check- if the new course of the ball also leads to a collision
            trajectory = calculateTrajectory(dt);
            collision = gameEnvironment.getClosestCollision(trajectory);
            //no collision expected at the new course OR the second collision is with the paddle(makes the ball stuck)
            if (collision == null || collision.collisionObject() instanceof Paddle) {
                this.point = this.velocity.applyToPoint(this.point, dt);
            } else { //collision is expected to occur with the new course
                //changes the ball course to the opposite of where it originally came from.
                this.velocity = new Velocity(previous.getDx() * -1, previous.getDy() * -1);
            }
        }
    }

    /**
     * Calculates the trajectory - "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.
     *
     * @param dt amount of seconds passed since the last call
     * @return trajectory line
     */
    public Line calculateTrajectory(double dt) {
        //default values of velocity if the velocity wasn't set before trying to move the ball.
        double dx = 1;
        double dy = 1;
        try {
            dx = this.velocity.getDx();
            dy = this.velocity.getDy();
            //if the velocity wasn't set before trying to move the ball, prints a message and sets the default values.
        } catch (NullPointerException e) {
            System.out.println("Ball's velocity wasn't defined. Velocity is now default values: dx=1, dy=1.");
            this.setVelocity(dx, dy);
        }
        //trajectory ends where the ball would advance to in its next step.
        Point trajectoryEnd = new Point(this.point.getX() + dx * dt, this.point.getY() + dy * dt);
        /* adjusting the trajectory to be longer so that the ball will move to "almost" the hit point, but just
         slightly before it */
        if (this.velocity.getDx() >= 0) {
            trajectoryEnd.setX(trajectoryEnd.getX() + this.size / 2);
        }
        if (this.velocity.getDx() < 0) {
            trajectoryEnd.setX(trajectoryEnd.getX() - this.size / 2);
        }
        if (this.velocity.getDy() >= 0) {
            trajectoryEnd.setY(trajectoryEnd.getY() + this.size / 2);
        }
        if (this.velocity.getDy() < 0) {
            trajectoryEnd.setY(trajectoryEnd.getY() - this.size / 2);
        }
        return new Line(this.point, trajectoryEnd);
    }

    /**
     * adds the ball to the game-as a sprite.
     * also, increases the number of balls in the game.
     *
     * @param g game
     */
    public void addToGame(GameLevel g) {
        g.addSprite(this);
        g.getNumOfBalls().increase(1);
    }

    /**
     * removers the ball from the game-as a sprite.
     * Decrease in numOfBalls is executed in BallsRemover.
     *
     * @param g game
     */
    public void removeFromGame(GameLevel g) {
        g.removeSprite(this);
    }
}
```

```java
package shapes;

/**
 * Classname: Line
 * A line (actually a line-segment) connects two points - a start point and an end point.
 * Lines have lengths, and may intersect with other lines.
 * It can also tell if it is the same as another line segment.
 *
 * @author Elad Israel
 * @version 3.0 20/05/2018
 */
public class Line {

    // Members - what defines a line
    private Point start;
    private Point end;
    private Point inter;

    /**
     * Constructor 1.
     * Constructs a Line using starting point and ending point.
     *
     * @param start starting point of this line.
     * @param end   ending point of this line.
     */
    public Line(Point start, Point end) {
        this.start = start;
        this.end = end;
    }

    /**
     * Constructor 2.
     * Constructs a Line using x coordinate and y coordinate of a starting point
     * and x coordinate and y coordinate of an ending point.
     *
     * @param x1 coordinate X of the starting point.
     * @param y1 coordinate Y of the starting point.
     * @param x2 coordinate X of the ending point.
     * @param y2 coordinate Y of the ending point.
     */

    public Line(double x1, double y1, double x2, double y2) {
        this.start = new Point(x1, y1);
        this.end = new Point(x2, y2);
    }

    /**
     * Return the length of the line.
     *
     * @return length
     */
    public double length() {
        return this.start.distance(this.end);
    }

    /**
     * Returns the middle point of the line.
     *
     * @return middle point
     */
    public Point middle() {
        double midX = (this.start.getX() + this.end.getX()) / 2;
        double midY = (this.start.getY() + this.end.getY()) / 2;
        return new Point(midX, midY);
    }

    /**
     * Returns the starting point of the line.
     *
     * @return start point
     */
    public Point start() {
        return this.start;
    }

    /**
     * Returns the starting point of the line.
     *
     * @return start point
     */
    public Point end() {
        return this.end;
    }

    /**
     * Returns true if the lines intersect (calculates the intersection point in the process),
     * and returns false otherwise:
     * if the lines have the same slope and if don't have the same slope but don't intersect.
     *
     * @param other other line.
     * @return true/false- intersects/not.
     */
    public boolean isIntersecting(Line other) {
        //stores the intersection point's coordinates.
        double interX;
        double interY;
        //stores the coordinates of this line and other line to avoid calling functions a lot and for readability.
        double thisStartX = this.start().getX();
        double thisStartY = this.start().getY();
        double thisEndX = this.end().getX();
        double thisEndY = this.end().getY();
        double otherStartX = other.start().getX();
        double otherStartY = other.start().getY();
        double otherEndX = other.end().getX();
        double otherEndY = other.end().getY();
        //stores the slopes of the two lines
        double thisSlope;
        double otherSlope;

        /*
        formulas explanation:

        definition:
        x1= x of the start of this line
        x2= x of the end of this line
        y1= y of the start of this line
        y2= y of the end of this line
        a1= x of the start of other line
        a2= x of the end of other line
        b1= y of the start of other line
        b2= y of the end of other line

        we have an equation for each line:
        first line(this line): y=m1(x-x1)+y1 when m1 can be: (y2-y1)/(x2-x1)
        second line(other line): y=m2(x-a1)+b1 when m2 can be: (b2-b1)/(a2-a1)
        */

        //at least one of the lines is vertical
        if (thisEndX - thisStartX == 0 || otherEndX - otherStartX == 0) {
            //both vertical- same slope(infinity)
            if ((thisEndX - thisStartX == 0) && (otherEndX - otherStartX == 0)) {
                return false;
            }
            // this line is vertical
            if (thisEndX - thisStartX == 0) {
                //m1=(y2-y1)-(x2-x1)
                otherSlope = (otherEndY - otherStartY) / (otherEndX - otherStartX);
                //X-coordinate of the intersection is any X of this line
                interX = this.start.getX();
                //intersection Y=m2*(x1-a1)+b1
                interY = otherSlope * (interX - otherStartX) + otherStartY;
                //other line is vertical
            } else {
                //m2=(b2-b1)/(a2-a1)
                thisSlope = (thisEndY - thisStartY) / (thisEndX - thisStartX);
                //X-coordinate of the intersection is any X of other line
                interX = other.start.getX();
                //intersection Y=m1*(b1-x1)+y1
                interY = thisSlope * (interX - thisStartX) + thisStartY;
            }
            //no lines are vertical- calculate slopes as explained above(regular formula).
        } else {
            thisSlope = (thisEndY - thisStartY) / (thisEndX - thisStartX);
            otherSlope = (otherEndY - otherStartY) / (otherEndX - otherStartX);

            //the lines are parallel
            if (thisSlope == otherSlope) {
                return false;
            }

            /*
             calculates the X and Y coordinates of the intersection
             X coordinate= (m2x2-y2-m2a1+b1)/(m2-m1) as can be calculated from the formulas above by comparing the Y
             of both equations and then isolating the x of the intersection point.
             Y coordinate= simply placing the X coordinate found in one of the formulas.
             */
            interX = (thisSlope * thisStartX - thisStartY - otherSlope * otherStartX + otherStartY)
                    / (thisSlope - otherSlope);
            interY = thisSlope * interX - thisSlope * thisStartX + thisStartY;
        }
        //creates a point for this intersection point.
        this.inter = new Point(interX, interY);

        //if intersection point is between the limits of the line-segments it's treated as an intersection point.
        return ((this.inter.getX() >= Math.min(thisStartX, thisEndX))
                && (this.inter.getX() <= Math.max(thisStartX, thisEndX))
                && (this.inter.getY() >= Math.min(thisStartY, thisEndY))
                && (this.inter.getY() <= Math.max(thisStartY, thisEndY))
                && (this.inter.getX() >= Math.min(otherStartX, otherEndX))
                && (this.inter.getX() <= Math.max(otherStartX, otherEndX))
                && (this.inter.getY() >= Math.min(otherStartY, otherEndY))
                && (this.inter.getY() <= Math.max(otherStartY, otherEndY)));
    }

    /**
     * Returns the intersection point if the lines intersect, and null otherwise.
     * uses isIntersecting for the calculation.
     *
     * @param other other line
     * @return intersection point if there is one, null otherwise.
     */
    public Point intersectionWith(Line other) {
        if (this.isIntersecting(other)) {
            return this.inter;
        } else {
            return null;
        }
    }

    /**
     * equals - return true if the lines are equal, false otherwise.
     * important! two line that whose starting and ending points are the similar but opposite- aren't equals!
     *
     * @param other other line
     * @return are equals or not(boolean)
     */
    public boolean equals(Line other) {
        return ((this.start.equals(other.start())) && (this.end.equals(other.end())));
    }

    /**
     * If this line does not intersect with the rectangle, return null.
     * Otherwise, return the closest intersection point to the
     * start of the line.
     *
     * @param rect rectangle to check intersections
     * @return closest intersection point to start of line
     */
    public Point closestIntersectionToStartOfLine(Rectangle rect) {
        java.util.List<Point> intersectionPArr = rect.intersectionPoints(this);
        //no intersection points
        if (intersectionPArr.size() == 0) {
            return null;
        }
        //one intersection point
        if (intersectionPArr.size() == 1) {
            return intersectionPArr.get(0);
        }
        //two intersection points - returns the closest one
        if (intersectionPArr.get(0).distance(this.start) < intersectionPArr.get(1).distance(this.start)) {
            return intersectionPArr.get(0);
        } else {
            return intersectionPArr.get(1);
        }
    }

    /**
     * checks whether a given point is on this line.
     *
     * @param checkedPoint given point to check
     * @return true if is, false if isn't
     */
    public boolean isPointOnTheLine(Point checkedPoint) {
        return (checkedPoint.distance(this.start) + checkedPoint.distance(this.end) == this.start.distance(this.end));
    }
}
```

```java
package shapes;

/**
 * Classname: Point
 * A point has an x and a y value, and can measure the distance to other points,
 * and if its is equal to another point.
 *
 * @author Elad Israel
 * @version 1.2 20/04/2018
 */
public class Point {

    // Members
    private double x;
    private double y;

    /**
     * Constructor.
     * Constructs a Point using x coordinate and y coordinate.
     *
     * @param x X coordinate of this point.
     * @param y Y coordinate of this point.
     */
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    /**
     * Access method- Return the x value of this point.
     *
     * @return x value of this point
     */
    public double getX() {
        return this.x;
    }

    /**
     * Setter for x.
     *
     * @param newX the new x.
     */
    public void setX(double newX) {
        this.x = newX;
    }

    /**
     * Access method- Return the y value of this point.
     *
     * @return y value of this point
     */
    public double getY() {
        return this.y;
    }

    /**
     * Setter for y.
     *
     * @param newY the new y.
     */
    public void setY(double newY) {
        this.y = newY;
    }


    /**
     * distance - return the distance of this point to the other point.
     *
     * @param other other point
     * @return distance
     */
    public double distance(Point other) {
        double dx = this.x - other.getX();
        double dy = this.y - other.getY();
        return Math.sqrt((dx * dx) + (dy * dy));
    }

    /**
     * equals - return true if the points are equal, false otherwise.
     *
     * @param other other point
     * @return are equals or not(boolean)
     */
    public boolean equals(Point other) {
        return ((this.x == other.getX()) && (this.y == other.getY()));
    }
}
```

```java
package shapes;

import biuoop.DrawSurface;

import java.awt.Color;
import java.util.ArrayList;


/**
 * Classname: Rectangle
 * A Rectangle has size, color, and location (a Point).
 * it also has edges and fill and draw colors.
 * Rectangles also know how to draw themselves on a DrawSurface.
 *
 * @author Elad Israel
 * @version 1.0 20/04/2018
 */
public class Rectangle {
    private Point upperLeft;
    private double width;
    private double height;
    private Line upperEdge;
    private Line lowerEdge;
    private Line leftEdge;
    private Line rightEdge;
    private java.awt.Color fillColor;
    private java.awt.Color drawColor;

    // Create a new rectangle with location and width/height.
    //and edges

    /**
     * Constructor1
     * construct a Rectangle using upper-left point ,width and height. and sets the edges.
     *
     * @param upperLeft upper-left corner
     * @param width     of the rectangle
     * @param height    of the rectangle
     */
    public Rectangle(Point upperLeft, double width, double height) {
        this.upperLeft = upperLeft;
        this.width = width;
        this.height = height;
        setEdges();
    }

    /**
     * Constructor2
     * construct a Rectangle using upper-left point ,width and height, and a color to fill. and sets the edges.
     *
     * @param upperLeft upper-left corner
     * @param width     of the rectangle
     * @param height    of the rectangle
     * @param fillColor of the rectangle.
     */
    public Rectangle(Point upperLeft, double width, double height, java.awt.Color fillColor) {
        this.upperLeft = upperLeft;
        this.width = width;
        this.height = height;
        setEdges();
        this.fillColor = fillColor;
    }

    /**
     * Constructor3
     * construct a Rectangle using upper-left point ,width and height, and fill and draw colors. and sets the edges.
     *
     * @param upperLeft upper-left corner
     * @param width     of the rectangle
     * @param height    of the rectangle
     * @param fillColor of the rectangle.
     * @param drawColor of the rectangle.
     */
    public Rectangle(Point upperLeft, double width, double height, java.awt.Color fillColor, java.awt.Color drawColor) {
        this.upperLeft = upperLeft;
        this.width = width;
        this.height = height;
        setEdges();
        this.fillColor = fillColor;
        this.drawColor = drawColor;
    }

    /**
     * sets the edges of the Rectangle accourding to the upper left corner received.
     */
    private void setEdges() {
        Point upperRight = new Point(this.upperLeft.getX() + this.width, this.upperLeft.getY());
        Point lowerLeft = new Point(this.upperLeft.getX(), this.upperLeft.getY() + this.getHeight());
        Point lowerRight = new Point(this.upperLeft.getX() + this.width, this.upperLeft.getY() + this.height);
        this.upperEdge = new Line(this.upperLeft, upperRight);
        this.lowerEdge = new Line(lowerLeft, lowerRight);
        this.leftEdge = new Line(this.upperLeft, lowerLeft);
        this.rightEdge = new Line(upperRight, lowerRight);
    }

    /**
     * Setter for the upperLeft point- change the rectangle position and reset the edges.
     *
     * @param newUpperLeft to set.
     */
    public void changePosition(Point newUpperLeft) {
        this.upperLeft = newUpperLeft;
        setEdges();
    }

    /**
     * Getter of the rectangle's width.
     *
     * @return the width of the rectangle
     */
    public double getWidth() {
        return this.width;
    }

    /**
     * Sets width.
     *
     * @param widthToSet the width
     */
    public void setWidth(double widthToSet) {
        this.width = widthToSet;
    }

    /**
     * Getter of the rectangle's height.
     *
     * @return the height of the rectangle
     */
    public double getHeight() {
        return this.height;
    }

    /**
     * Sets height.
     *
     * @param heightToSet the height
     */
    public void setHeight(double heightToSet) {
        this.height = heightToSet;
    }

    /**
     * Getter of the upper-left point of the rectangle.
     *
     * @return the upper-left point of the rectangle.
     */
    public Point getUpperLeft() {
        return this.upperLeft;
    }

    /**
     * Sets upper left.
     *
     * @param upperLeftToSet the upper left
     */
    public void setUpperLeft(Point upperLeftToSet) {
        this.upperLeft = upperLeftToSet;
    }

    // Returns the upper-left point of the rectangle.

    /**
     * Gets the lower edge(line) of the rectangle.
     *
     * @return lower line
     */
    public Line getLowerEdge() {
        return this.lowerEdge;
    }

    /**
     * Gets the upper edge(line) of the rectangle.
     *
     * @return upper line
     */
    public Line getUpperEdge() {
        return this.upperEdge;
    }

    /**
```

```java
     * Gets the left edge(line) of the rectangle.
     *
     * @return left line
     */
    public Line getLeftEdge() {
        return this.leftEdge;
    }

    /**
     * Gets the right edge(line) of the rectangle.
     *
     * @return right line
     */
    public Line getRightEdge() {
        return this.rightEdge;
    }

    /**
     * Access method- Return the fill color of this rectangle.
     *
     * @return the fill color of this rectangle.
     */
    public java.awt.Color getFillColor() {
        return this.fillColor;
    }

    /**
     * Access method- Return the draw color of this rectangle.
     *
     * @return the draw color of this rectangle.
     */
    public java.awt.Color getDrawColor() {
        return this.drawColor;
    }

    /**
     * Sets draw color.
     *
     * @param drawColorToSet the draw color
     */
    public void setDrawColor(Color drawColorToSet) {
        this.drawColor = drawColorToSet;
    }

    /**
     * draws this Rectangle on the given DrawSurface.
     *
     * @param surface drawSurface
     */
    public void drawOn(DrawSurface surface) {

        //default colors if no color was entered
        if (this.fillColor == null) {
            this.fillColor = Color.black;

        }
        if (this.drawColor == null) {
            this.drawColor = Color.black;
        }

        surface.setColor(this.fillColor);
        surface.fillRectangle((int) this.upperLeft.getX(), (int) this.upperLeft.getY(),
                (int) this.width, (int) this.height);
        surface.setColor(this.drawColor);
        surface.drawRectangle((int) this.upperLeft.getX(), (int) this.upperLeft.getY(),
                (int) this.width, (int) this.height);
    }

    /**
     * Return a (possibly empty) List of intersection points of the rectangle with the specified line.
     *
     * @param line the line to check with.
     * @return list of intersection points
     */
    public java.util.List<Point> intersectionPoints(Line line) {
        java.util.List<Point> intersectionPArr = new ArrayList<>();
        if (this.upperEdge.isIntersecting(line)) {
            intersectionPArr.add(this.upperEdge.intersectionWith(line));
        }
        if (this.lowerEdge.isIntersecting(line)) {
            intersectionPArr.add(this.lowerEdge.intersectionWith(line));
        }
        if (this.leftEdge.isIntersecting(line)) {
            intersectionPArr.add(this.leftEdge.intersectionWith(line));
        }
        if (this.rightEdge.isIntersecting(line)) {
            intersectionPArr.add(this.rightEdge.intersectionWith(line));
        }
        return intersectionPArr;
    }
```

```java
package animation;

/**
 * menu interface
 * When the game starts, the user will see a screen stating the game name (Arkanoid), and a list
 * of several options of what to do next.
 *
 * @param <T> the type parameter
 * @author Elad Israel
 * @version 4.0 17/06/2018
 */

public interface Menu<T> extends Animation {

    /**
     * Add selection to the menu.
     *
     * @param key       the key
     * @param message   the message
     * @param returnVal the return val
     */
    void addSelection(String key, String message, T returnVal);

    /**
     * Gets status.
     *
     * @return the status
     */
    T getStatus();

    /**
     * Add sub menu.
     *
     * @param key     the key
     * @param message the message
     * @param subMenu the sub menu
     */
    void addSubMenu(String key, String message, Menu<T> subMenu);
}
```

```java
package animation;

/**
 * The interface Task.
 *
 * @param <T> the type parameter
 */
public interface Task<T> {
    /**
     * runs the task.
     *
     * @return T
     */
    T run();
}
```

```java
package animation;

import biuoop.DrawSurface;


/**
 * interface name: Animation
 * The Animation interface.
 * describes an animation object-
 * any animation should specify what to do in each frame, and notify when to stop the animation.
 *
 * @author Elad Israel
 * @version 4.0 17/06/2018
 */
public interface Animation {
    /**
     * Do one frame of the animation.
     *
     * @param d the draw surface
     * @param dt amount of seconds passed since the last call
     */
    void doOneFrame(DrawSurface d, double dt);

    /**
     * Should the animation stop.
     *
     * @return boolean
     */
    boolean shouldStop();
}
```

```java
package animation;

import biuoop.DrawSurface;
import biuoop.KeyboardSensor;
import game.Counter;

import java.awt.Color;

/**
 * Classname: EndScreen.
 * Once the game is over (either the player run out of lives or managed to clear all the levels),
 * we will display the final score. If the game ended with the player losing all his lives,
 * the end screen should display the message "Game Over. Your score is X" (X being the final score).
 * If the game ended by clearing all the levels, the screen should display "You Win! Your score is X".
 * The "end screen" should persist until the space key is pressed.
 * After the space key is pressed, your program should terminate.
 *
 * @author Elad Israel
 * @version 4.0 17/06/2018
 */
public class EndScreen implements Animation {
    private Counter score;
    private boolean stop;
    private boolean won;

    /**
     * Instantiates a new End screen.
     *
     * @param k     the KeyboardSensor
     * @param score the score to display
     * @param won   did the player won or lost
     */
    public EndScreen(KeyboardSensor k, Counter score, boolean won) {
        this.stop = false;
        this.score = score;
        this.won = won;
    }

    /**
     * Do one frame of the animation.
     *
     * @param d the draw surface
     * @param dt amount of seconds passed since the last call
     */
    public void doOneFrame(DrawSurface d, double dt) {
        if (this.won) {
            d.setColor(Color.blue);
            d.drawText(10, d.getHeight() / 2, "You Win! Your score is " + this.score.getValue(), 40);
        } else {
            d.setColor(Color.red);
            d.drawText(10, d.getHeight() / 2, "Game Over. Your score is " + this.score.getValue(), 40);
        }
    }

    /**
     * Should the animation stop.
     *
     * @return boolean
     */
    public boolean shouldStop() {
        return this.stop;
    }
}
```

```java
package animation;

import biuoop.DrawSurface;
import biuoop.KeyboardSensor;
import game.GameEnvironment;
import game.Paddle;
import game.SpriteCollection;
import game.Counter;
import game.Collidable;
import game.Sprite;
import game.Block;
import game.ScoreIndicator;
import game.LivesIndicator;
import game.NameOfLevelIndicator;
import game.Velocity;
import levels.LevelInformation;
import listeners.BallRemover;
import listeners.BlockRemover;
import listeners.ScoreTrackingListener;
import shapes.Ball;
import shapes.Point;
import shapes.Rectangle;

import java.awt.Color;
import java.util.ArrayList;

/**
 * Class name: GameLevel
 * A class that will hold the sprites and the collidables, and will be in charge of the animation.
 *
 * @author Elad Israel
 * @version 4.0 17/06/2018
 */
public class GameLevel implements Animation {

    private static final int UP_AND_DOWN_FRAMES_HEIGHT = 25;
    private static final int LEFT_AND_RIGHT_FRAMES_WIDTH = 25;
    private static final java.awt.Color FRAMES_COLOR = Color.gray;
    private final int frameHeight;
    private final int frameWidth;
    private SpriteCollection sprites;
    private GameEnvironment environment;
    private Paddle paddle;
    private KeyboardSensor keyboardSensor;
    private Counter numOfBlocks;
    private BlockRemover blockRemover;
    private Counter numOfBalls;
    private BallRemover ballRemover;
    private Counter score;
    private Counter numOfLives;
    private AnimationRunner runner;
    private boolean running;
    private LevelInformation levelInformation;
    private Counter blocksLeftToRemove;


    /**
     * Constructor- creates the sprite collection, environment, and keyboard sensor of the game.
     *
     * @param levelInformation the level information
     * @param keyboardSensor   the keyboard sensor
     * @param animationRunner  the animation runner
     * @param score            the score
     * @param numOfLives       the num of lives
     * @param frameWidth       the frame width
     * @param frameHeight      the frame height
     */
    public GameLevel(LevelInformation levelInformation, KeyboardSensor keyboardSensor, AnimationRunner animationRunner,
                     Counter score, Counter numOfLives, final int frameWidth, final int frameHeight) {
        this.frameWidth = frameWidth;
        this.frameHeight = frameHeight;
        this.runner = animationRunner;
        this.keyboardSensor = keyboardSensor;
        this.sprites = new SpriteCollection();
        this.environment = new GameEnvironment(new ArrayList<>());
        this.numOfBlocks = new Counter();
        this.numOfBalls = new Counter();
        this.ballRemover = new BallRemover(this, this.numOfBalls);
        this.levelInformation = levelInformation;
        this.blocksLeftToRemove = new Counter();
        this.blocksLeftToRemove.increase(this.levelInformation.numberOfBlocksToRemove());
        this.blockRemover = new BlockRemover(this, this.numOfBlocks, this.blocksLeftToRemove);
        this.score = score;
        this.numOfLives = numOfLives;
    }

    /**
     * add the given collidable to the collidables collection in the environment.
     *
     * @param c given collidable.
     */
    public void addCollidable(Collidable c) {
        try {
            environment.addCollidable(c);
        } catch (RuntimeException nullPointer) {
            throw new RuntimeException("Collidable field wasn't initialized!");
        }
    }

    /**
     * removes the given collidable from the collidables collection in the environment.
     *
     * @param c given collidable.
     */
    public void removeCollidable(Collidable c) {
        try {
            environment.removeCollidable(c);
        } catch (RuntimeException nullPointer) {
            throw new RuntimeException("Collidable field wasn't initialized!");
        }
    }

    /**
     * add the given sprite to the sprite collection.
     *
     * @param s given sprite.
     */
    public void addSprite(Sprite s) {
        sprites.addSprite(s);
    }

    /**
     * removes the given sprite from the sprite collection.
     *
     * @param s given sprite.
     */
    public void removeSprite(Sprite s) {
        sprites.removeSprite(s);
    }


    /**
     * Initialize a new game: create the Blocks and Ball (and Paddle) and add them to the game.
     */
    public void initialize() {
        addSprite(this.levelInformation.getBackground());
        initializeFrames();
        initializeCenterBlocks();
    }


    /**
     * creates frames from all sides to prevent the balls of leaving the screen.
     */
    public void initializeFrames() {
        Block right = new Block(frameWidth - LEFT_AND_RIGHT_FRAMES_WIDTH, 0, LEFT_AND_RIGHT_FRAMES_WIDTH,
                frameHeight, FRAMES_COLOR, FRAMES_COLOR, 1);
        right.addToGame(this);
        Block left = new Block(0, 0, LEFT_AND_RIGHT_FRAMES_WIDTH, frameHeight, FRAMES_COLOR, FRAMES_COLOR, 1);
        left.addToGame(this);
        Block up = new Block(0, UP_AND_DOWN_FRAMES_HEIGHT, frameWidth, UP_AND_DOWN_FRAMES_HEIGHT, FRAMES_COLOR,
                FRAMES_COLOR, 1);
        up.addToGame(this);
        //"death region". lowered beneath the gui so that the balls will disappear after leaving the screen.
        Block down = new Block(-frameWidth, frameHeight + UP_AND_DOWN_FRAMES_HEIGHT, frameWidth * 3,
                UP_AND_DOWN_FRAMES_HEIGHT, FRAMES_COLOR, FRAMES_COLOR, 1);
        down.addToGame(this);
        down.addHitListener(this.ballRemover);

        this.getNumOfBlocks().decrease(4);

        //initialize score sprite
        ScoreIndicator scoreIndicator = new ScoreIndicator(new Rectangle(new Point(0, 0), frameWidth,
                UP_AND_DOWN_FRAMES_HEIGHT, Color.white, Color.white), this.score);
        scoreIndicator.addToGame(this);

        LivesIndicator livesIndicator = new LivesIndicator(this.numOfLives);
        livesIndicator.addToGame(this);

        NameOfLevelIndicator nameOfLevelIndicator = new NameOfLevelIndicator(this.levelInformation.levelName());
        nameOfLevelIndicator.addToGame(this);
    }

    /**
     * Creates the blocks in the center of the screen- the ones the ball will collide with and destroy.
     */
    public void initializeCenterBlocks() {
        ScoreTrackingListener scoreTrackingListener = new ScoreTrackingListener(this.score);

        for (Block block : this.levelInformation.blocks()) {
            block.addToGame(this);
            block.addHitListener(this.blockRemover);
            block.addHitListener(scoreTrackingListener);
        }
    }


    /**
     * Creates two balls that will bounce around the screen, and the paddle that will try to prevent them from
     * falling down(by the user).
     *
     * @return paddle to remove by playOneTurn
     */
    public Paddle initializeBallsAndPaddle() {
        final int paddleHeight = 15;
        final int paddleWidth = this.levelInformation.paddleWidth();
        //Paddle
        this.paddle = new Paddle(new Point(frameWidth / 2 - paddleWidth / 2,
                frameHeight - UP_AND_DOWN_FRAMES_HEIGHT), paddleWidth, paddleHeight,
                this.levelInformation.paddleSpeed(), Color.yellow, java.awt.Color.black, keyboardSensor);
        this.paddle.addToGame(this);

        //balls
        for (Velocity velocityOfBall : this.levelInformation.initialBallVelocities()) {
            double ballX = this.paddle.getCollisionRectangle().getUpperLeft().getX()
                    + this.levelInformation.paddleWidth() / 2;
            double ballY = this.paddle.getCollisionRectangle().getUpperLeft().getY() - 15;
            Ball ball = new Ball((int) ballX, (int) ballY, 5, Color.white, Color.black);
            ball.setGameEnvironment(this.environment);
            ball.setVelocity(velocityOfBall);
            ball.addToGame(this);
        }

        return paddle;
    }

    /**
     * Should the animation stop.
     *
     * @return boolean
     */
    public boolean shouldStop() {
        return !this.running;
    }

    /**
     * Do one frame of the animation.
     *
     * @param d the draw surface
     * @param dt amount of seconds passed since the last call
     */
    public void doOneFrame(DrawSurface d, double dt) {
        if (this.keyboardSensor.isPressed("p")) {
            this.runner.run(new KeyPressStoppableAnimation(this.keyboardSensor, "space",
                    new PauseScreen(this.keyboardSensor)));
        }
        this.sprites.drawAllOn(d);
        this.sprites.notifyAllTimePassed(dt);
        if (this.numOfBlocks.getValue() == 0 || this.blocksLeftToRemove.getValue() <= 0) {
            paddle.removeFromGame(this);
            this.score.increase(100);
            this.running = false;
        }
        if (this.numOfBalls.getValue() == 0) {
            paddle.removeFromGame(this);
            this.numOfLives.decrease(1);
            this.running = false;
        }
    }

    /**
     * playing one turn.
     * playOneTurn starts by creating balls and putting the paddle at the bottom of the screen.
     */
    public void playOneTurn() {
        this.paddle = initializeBallsAndPaddle();
        this.runner.run(new CountdownAnimation(2, 3, this.sprites)); // countdown before turn starts.
        this.running = true;
        // use our runner to run the current animation -- which is one turn of the game.
        this.runner.run(this);
    }

    /**
     * Gets num of blocks.
     *
     * @return the num of blocks
     */
    public Counter getNumOfBlocks() {
        return this.numOfBlocks;
    }
    /**
     * Gets blocks left to remove.
     *
     * @return the blocks left to remove
     */
    public Counter getBlocksLeftToRemove() {
        return this.blocksLeftToRemove;
    }

    /**
     * Gets num of balls.
     *
     * @return the num of balls
     */
    public Counter getNumOfBalls() {
        return this.numOfBalls;
    }

    /**
     * Gets num of lives.
     *
     * @return the num of lives
     */
    public Counter getNumOfLives() {
        return this.numOfLives;
    }
}
```

```java
package animation;

import biuoop.DrawSurface;
import biuoop.KeyboardSensor;

import java.awt.Color;

/**
 * Classname: PauseScreen.
 * Display a screen with the message paused -- press space to continue until a key is pressed.
 * An option to pause the game when pressing the p key.
 *
 * @author Elad Israel
 * @version 3.0 20/05/2018
 */
public class PauseScreen implements Animation {
    private KeyboardSensor keyboard;
    private boolean stop;

    /**
     * Constructor.
     *
     * @param k the KeyboardSensor.
     */
    public PauseScreen(KeyboardSensor k) {
        this.keyboard = k;
        this.stop = false;
    }

    /**
     * Do one frame of the animation.
     *
     * @param d  the draw surface
     * @param dt amount of seconds passed since the last call
     */
    public void doOneFrame(DrawSurface d, double dt) {
        d.setColor(Color.black);
        d.drawText(10, d.getHeight() / 2, "paused -- press space to continue", 32);
    }

    /**
     * Should the animation stop.
     *
     * @return boolean
     */
    public boolean shouldStop() {
        return this.stop;
    }
}
```

```java
package animation;

import biuoop.DrawSurface;
import biuoop.KeyboardSensor;

import java.awt.Color;
import java.util.ArrayList;
import java.util.List;


/**
 * Our Menu will need to be displayed on screen, so it will be an Animation. Unlike the other animation loops we had,
 * this one will need to return a value when it is done. We may want to add a nice background to our menu. For this, we
 * will provide it with a method that will accept a background sprite and display it.
 *
 * @param <T> the type parameter
 * @author Elad Israel
 * @version 4.0 17/06/2018
 */
public class MenuAnimation<T> implements Menu<T> {


    private List<String> keys;
    private List<String> messages;
    private List<T> returnVals;
    private String title;
    private KeyboardSensor keyboard;
    private boolean stop;
    private T status;
    private AnimationRunner animationRunner;
    private List<Menu<T>> subMenus;
    private List<Boolean> isSubMenu;


    /**
     * Instantiates a new Menu animation.
     *
     * @param title           the title
     * @param keyboard        the keyboard
     * @param animationRunner the animation runner
     */
    public MenuAnimation(String title, KeyboardSensor keyboard, AnimationRunner animationRunner) {
        this.keys = new ArrayList<String>();
        this.messages = new ArrayList<String>();
        this.returnVals = new ArrayList<T>();
        this.title = title;
        this.keyboard = keyboard;
        this.stop = false;
        this.animationRunner = animationRunner;
        this.subMenus = new ArrayList<>();
        this.isSubMenu = new ArrayList<>();
    }

    /**
     * Add selection to the menu.
     *
     * @param key       the key
     * @param message   the message
     * @param returnVal the return val
     */
    public void addSelection(String key, String message, T returnVal) {
        this.keys.add(key);
        this.messages.add(message);
        this.returnVals.add(returnVal);
        this.subMenus.add(null);
        this.isSubMenu.add(false);
    }

    /**
     * Gets status.
     *
     * @return the status
     */
    public T getStatus() {
        if (this.status == null) {
            throw new RuntimeException("status wasn't initialized");
        }
        T tempStatus = this.status;
        //reset fields
        this.status = null;
        this.stop = false;
        return tempStatus;
    }

    /**
     * Do one frame of the animation.
     *
     * @param d the draw surface
     * @param dt amount of seconds passed since the last call
     */
    public void doOneFrame(DrawSurface d, double dt) {
        d.setColor(Color.gray.darker().darker());
        d.fillRectangle(0, 0, d.getWidth(), d.getHeight());
        d.setColor(Color.YELLOW);
        d.drawText(50, 50, this.title, 50);
        d.setColor(Color.WHITE);
        for (int i = 0; i < this.keys.size(); i++) {
            d.drawText(100, 150 + i * 50, "(" + this.keys.get(i) + ") " + this.messages.get(i), 32);
        }
        for (int i = 0; i < this.keys.size(); i++) {
            if (this.keyboard.isPressed(this.keys.get(i))) {
                if (!this.isSubMenu.get(i)) {
                    this.status = this.returnVals.get(i);
                    this.stop = true;
                    break;
                } else {
                    Menu<T> subMenu = this.subMenus.get(i);
                    this.animationRunner.run(subMenu);
                    this.status = subMenu.getStatus();
                    this.stop = true;
                    break;
                }
            }
        }
    }

    /**
     * Should the animation stop.
     *
     * @return boolean
     */
    public boolean shouldStop() {
        return this.stop;
    }

    /**
     * Add sub menu.
     *
     * @param key     the key
     * @param message the message
     * @param subMenu the sub menu
     */
    public void addSubMenu(String key, String message, Menu<T> subMenu) {
        this.subMenus.add(subMenu);
        this.keys.add(key);
        this.messages.add(message);
        this.returnVals.add(null);
        this.isSubMenu.add(true);
    }
}
```

```java
package animation;

import biuoop.DrawSurface;
import biuoop.GUI;
import biuoop.Sleeper;

/**
 * class name: AnimationRunner
 * The AnimationRunner takes an Animation object and runs it.
 *
 * @author Elad Israel
 * @version 4.0 17/06/2018
 */
public class AnimationRunner {
    private GUI gui;
    private int framesPerSecond;
    private Sleeper sleeper;

    /**
     * constructor.
     *
     * @param gui the graphical user interface of the game.
     */
    public AnimationRunner(GUI gui) {
        this.gui = gui;
        this.framesPerSecond = 60;
        this.sleeper = new Sleeper();
    }

    /**
     * run the animation.
     *
     * @param animation to run
     */
    public void run(Animation animation) {
        long millisecondsPerFrame = (long) (1000 / framesPerSecond);
        long timeAfterOneFrame = System.currentTimeMillis();
        while (true) {
            /*the time it takes to perform each loop may be non-negligible.
             We therefor subtract the time it takes to do the work from
             the sleep time of millisecondsPerFrame milliseconds.
            */
            long startTime = System.currentTimeMillis(); // timing
            DrawSurface d = this.gui.getDrawSurface();

            double dt = (System.currentTimeMillis() - timeAfterOneFrame) / 1000.0;
            animation.doOneFrame(d, dt);
            timeAfterOneFrame = System.currentTimeMillis();

            if (animation.shouldStop()) {
                return;
            }

            gui.show(d);
            long usedTime = System.currentTimeMillis() - startTime; //the time it took
            long milliSecondLeftToSleep = millisecondsPerFrame - usedTime; //time left to sleep after the iteration.
            if (milliSecondLeftToSleep > 0) { // there is still time to sleep
                sleeper.sleepFor(milliSecondLeftToSleep);
            }
        }
    }

    /**
     * Get gui gui.
     *
     * @return the gui
     */
    public GUI getGui() {
        return this.gui;
    }
}
```

```java
package animation;

import biuoop.DrawSurface;
import biuoop.Sleeper;
import game.SpriteCollection;

import java.awt.Color;


/**
 * Classname: CountdownAnimation.
 * The CountdownAnimation will display the given gameScreen,
 * for numOfSeconds seconds, and on top of them it will show
 * a countdown from countFrom back to 1, where each number will
 * appear on the screen for (numOfSeconds / countFrom) secods, before
 * it is replaced with the next one.
 *
 * @author Elad Israel
 * @version 4.0 17/06/2018
 */
public class CountdownAnimation implements Animation {
    private double numOfSeconds;
    private int countFrom;
    private int currentCount;
    private SpriteCollection gameScreen;
    private boolean stop;
    private Sleeper sleeper;

    /**
     * Constructor.
     *
     * @param numOfSeconds the num of seconds to delay
     * @param countFrom    count from this number
     * @param gameScreen   the game screen
     */
    public CountdownAnimation(double numOfSeconds, int countFrom, SpriteCollection gameScreen) {
        this.numOfSeconds = numOfSeconds;
        this.countFrom = countFrom;
        this.currentCount = countFrom;
        this.gameScreen = gameScreen;
        this.stop = false;
        this.sleeper = new Sleeper();
    }

    /**
     * Do one frame of the animation.
     *
     * @param d  the draw surface
     * @param dt amount of seconds passed since the last call
     */
    public void doOneFrame(DrawSurface d, double dt) {
        this.gameScreen.drawAllOn(d);
        //when count reaches 0 it shouldn't draw 0 on the screen.
        if (this.currentCount > 0) {
            d.setColor(Color.decode("#1B76F2"));
            d.drawText((int) (d.getWidth() / 2.05), d.getHeight() / 2, Integer.toString(this.currentCount), 50);
        }
        //not the first time(first time shouldn't sleep because gui wasn't shown yet.
        if (this.currentCount != this.countFrom) {
            this.sleeper.sleepFor((long) ((this.numOfSeconds / this.countFrom) * 1000));
        }
        this.currentCount--;
    }

    /**
     * Should the animation stop.
     *
     * @return boolean
     */
    public boolean shouldStop() {
        //count is over
        if (this.currentCount < 0) {
            return true;
        }
        return this.stop;
    }
}
```

```java
package animation;

import biuoop.DrawSurface;
import biuoop.KeyboardSensor;
import game.HighScoresTable;

import java.awt.Color;

/**
 * The High scores animation.
 * @author Elad Israel
 * @version 4.0 17/06/2018
 */
public class HighScoresAnimation implements Animation {

    private HighScoresTable scores;
    private String endKey;
    private boolean stop;
    private KeyboardSensor keyboard;


    /**
     * Instantiates a new High scores animation.
     *
     * @param scores   the scores
     * @param endKey   the end key
     * @param keyboard the keyboard
     */
    public HighScoresAnimation(HighScoresTable scores, String endKey, KeyboardSensor keyboard) {

        this.scores = scores;
        this.endKey = endKey;
        this.keyboard = keyboard;
        this.stop = false;
    }

    /**
     * Instantiates a new High scores animation.
     *
     * @param scores   the scores
     * @param keyboard the keyboard
     */
    public HighScoresAnimation(HighScoresTable scores, KeyboardSensor keyboard) {
        this.scores = scores;
        this.keyboard = keyboard;
        this.stop = false;
    }


    /**
     * Do one frame of the animation.
     *
     * @param d  the draw surface
     * @param dt amount of seconds passed since the last call
     */
    public void doOneFrame(DrawSurface d, double dt) {
        d.setColor(Color.gray);
        d.fillRectangle(0, 0, d.getWidth(), d.getHeight());
        d.setColor(Color.YELLOW);
        d.drawText(50, 50, "High Scores:", 50);

        d.setColor(Color.WHITE);
        d.drawText(100, 150, "Player Name", 32);
        d.setColor(Color.WHITE);
        d.drawText(500, 150, "Score", 32);
        d.drawText(100, 150, "____________________________________", 32);

        for (int i = 0; i < this.scores.getHighScores().size(); i++) {
            d.setColor(Color.BLUE);
            d.drawText(100, 200 + i * 50, this.scores.getHighScores().get(i).getName(), 32);
            d.setColor(Color.BLUE);
            d.drawText(500, 200 + i * 50, "" + this.scores.getHighScores().get(i).getScore(), 32);
        }
        d.setColor(Color.BLACK);
        d.drawText(200, 500, "Press space to continue", 32);
    }

    /**
     * Should the animation stop.
     *
     * @return boolean
     */
    public boolean shouldStop() {
        return this.stop;
    }
}
```

```java
package animation;

import biuoop.DrawSurface;
import biuoop.KeyboardSensor;

/**
 * wrap an existing animation and add a "waiting-for-key" behavior to it.
 *
 * @author Elad Israel
 * @version 4.0 17/06/2018
 */
public class KeyPressStoppableAnimation implements Animation {
    private Animation decoratedAnimation;
    private KeyboardSensor sensor;
    private String key;
    private boolean stop;
    private boolean isAlreadyPressed;

    /**
     * wrap an existing animation and add a "waiting-for-key" behavior to it.
     *
     * @param sensor    the sensor
     * @param key       the key
     * @param animation the animation
     */
    public KeyPressStoppableAnimation(KeyboardSensor sensor, String key, Animation animation) {
        this.decoratedAnimation = animation;
        this.sensor = sensor;
        this.key = key;
        this.stop = false;
        this.isAlreadyPressed = true;
    }

    /**
     * Do one frame of the animation.
     *
     * @param d the draw surface
     * @param dt amount of seconds passed since the last call
     */
    public void doOneFrame(DrawSurface d, double dt) {
        this.stop = false;
        if (this.sensor.isPressed(key)) {
            //the key was pressed before the animation started - ignore the key press
            if (this.isAlreadyPressed) {
                return;
            }
            this.stop = true;
        }
        this.isAlreadyPressed = false;
        this.decoratedAnimation.doOneFrame(d, dt);
    }

    /**
     * Should the animation stop.
     *
     * @return boolean
     */
    public boolean shouldStop() {
        return this.stop;
    }
}
```

```java
package listeners;

import animation.GameLevel;
import game.Block;
import game.Counter;
import shapes.Ball;


/**
 * class name: BallRemover
 * BallRemover is in charge of removing balls from the gameLevel, as well as keeping count
 * of the number of balls that remain.
 *
 * @author Elad Israel
 * @version 3.0 20/05/2018
 */
public class BallRemover implements HitListener {
    private GameLevel gameLevel;
    private Counter remainingBalls;

    /**
     * constructor.
     *
     * @param gameLevel      the game level
     * @param remainingBalls the remaining balls
     */
    public BallRemover(GameLevel gameLevel, Counter remainingBalls) {
        this.gameLevel = gameLevel;
        this.remainingBalls = remainingBalls;
    }

    /**
     * whenever a special block that will sit at (or slightly below) the bottom of the screen is hit,
     * it will function as a "death region".
     * the BallRemover is registered as a listener of the death-region block, so that BallRemover will be
     * notified whenever a ball hits the death-region. Whenever this happens, the BallRemover will remove the ball
     * from the gameLevel and update the balls counter.
     *
     * @param beingHit the death region block
     * @param hitter   the ball that hits the block
     */
    public void hitEvent(Block beingHit, Ball hitter) {
        hitter.removeFromGame(this.gameLevel);
        remainingBalls.decrease(1);
    }
}
```

```java
package listeners;

import game.Block;
import shapes.Ball;

/**
 * interface name: HitListener
 * Objects that want to be notified of hit events, should implement the HitListener interface,
 * and register themselves with a HitNotifier object using its addHitListener method.
 *
 * @author Elad Israel
 * @version 3.0 20/05/2018
 */
public interface HitListener {
    /**
     * This method is called whenever the beingHit object is hit.
     * The hitter parameter is the Ball that's doing the hitting.
     *
     * @param beingHit the object that is being hit.
     * @param hitter   the object that hit.
     */
    void hitEvent(Block beingHit, Ball hitter);
}
```

```java
package listeners;

/**
 * interface name: HitNotifier
 * The HitNotifier interface indicate that objects that implement it send notifications when they are being hit.
 *
 * @author Elad Israel
 * @version 3.0 20/05/2018
 */
public interface HitNotifier {

    /**
     * Add hl as a listener to hit events.
     *
     * @param hl HitListener to remove
     */
    void addHitListener(HitListener hl);

    /**
     * Remove hl from the list of listeners to hit events.
     *
     * @param hl HitListener to remove
     */
    void removeHitListener(HitListener hl);
}
```

```java
package listeners;

import animation.GameLevel;
import game.Block;
import game.Counter;
import shapes.Ball;

/**
 * Classname: BlockRemover.
 * a BlockRemover is in charge of removing blocks from the gameLevel, as well as keeping count
 * of the number of blocks that remain.
 *
 * @author Elad Israel
 * @version 3.0 20/05/2018
 */
public class BlockRemover implements HitListener {
    private GameLevel gameLevel;
    private Counter remainingBlocks;
    private Counter blocksLeftToRemove;


    /**
     * Constructor.
     *
     * @param gameLevel          the game level
     * @param remainingBlocks    the remaining blocks
     * @param blocksLeftToRemove the blocks left to remove
     */
    public BlockRemover(GameLevel gameLevel, Counter remainingBlocks, Counter blocksLeftToRemove) {
        this.gameLevel = gameLevel;
        this.remainingBlocks = remainingBlocks;
        this.blocksLeftToRemove = blocksLeftToRemove;
    }

    /**
     * Blocks that are hit and reach 0 hit-points should be removed
     * from the gameLevel.
     *
     * @param beingHit the block that was hit.
     * @param hitter   the ball that hit.
     **/
    public void hitEvent(Block beingHit, Ball hitter) {
        if (beingHit.getHitPoints() == 1) {
            beingHit.removeHitListener(this);
            beingHit.removeFromGame(this.gameLevel);
            remainingBlocks.decrease(1);
            blocksLeftToRemove.decrease(1);
        }
    }
}
```

```java
package listeners;

import game.Block;
import game.Counter;
import shapes.Ball;

/**
 * Class name: ScoreTrackingListener
 * updates the score counter when blocks are being hit and removed.
 *
 * @author Elad Israel
 * @version 3.0 20/05/2018
 */
public class ScoreTrackingListener implements HitListener {
    private Counter currentScore;

    /**
     * Constructor.
     *
     * @param scoreCounter the score counter
     */
    public ScoreTrackingListener(Counter scoreCounter) {
        this.currentScore = scoreCounter;
    }

    /**
     * This method is called whenever the beingHit object is hit.
     * The hitter parameter is the Ball that's doing the hitting.
     *
     * @param beingHit the object that is being hit.
     * @param hitter   the object that hit.
     */
    public void hitEvent(Block beingHit, Ball hitter) {
        if (beingHit.getHitPoints() > 1) {
            this.currentScore.increase(5);
        } else {
            this.currentScore.increase(10);
        }
    }
}
```