Campbell Phalen: Code Sample

Breakout

My final project for CIS 120 (the course for which I now am a teaching assistant) was to make a game using Java Swing (a library for designing GUI applications in Java). The game I made was Breakout (or Brickbreaker) with a couple of additional features.

Features

Apart from normal Breakout this project supports a few additional features. First and foremost, you can serailize the game state (using Java text serialization) so that you can save a game and load it back in later. Second, there are many different types of bricks that vary up the way the game can be played (for more information on the different brick types check out **BrickType.java**). Lastly, there are options for pausing, resetting, and re-randomizing the game state (this you can checkout by looking at the `Menu` object and associated callback methods in **Game.java**).

File Layout

Java is great for GUI programs (especially 2D games) because they lend themselves nicely to Object-Oriented programs, that's reflected here in the code structure. I have four main game objects each of which has private state and a `draw` function describing how they will be drawn to the screen

- **Ball.java**: This class contains the code for operations moving the ball and checking for collision between the ball and other game objects.
- **Paddle.java**: This class contains the code for operations and state necessary to move the paddle around the screen.
- **Wall.java**: This class contains all of the bricks in a 2D array which manages the collision of the ball with the individual bricks.
- **Brick.java**: This class is the actual individual brick and mostly is in charge of saving their individual state.

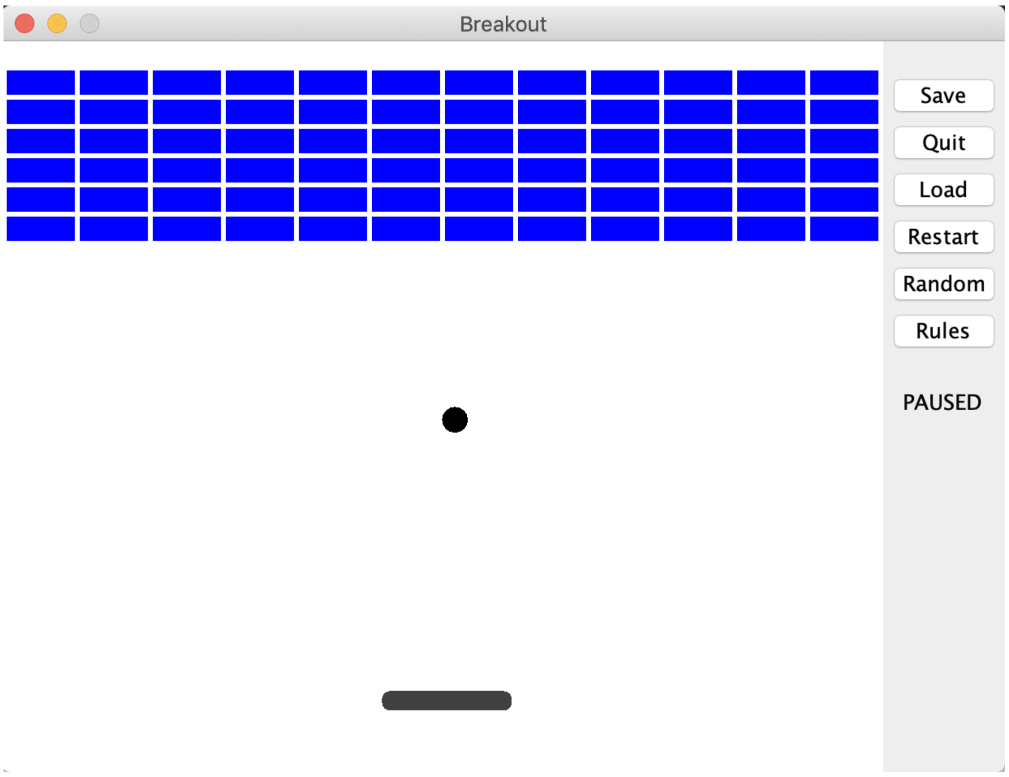
Then, there is the **Game.java** which is essentially the main loop for our program, containing the `public static void main(String[] args)` method which initializes the game loop and instantiates all other game objects. Lastly, there are two enumeration classes

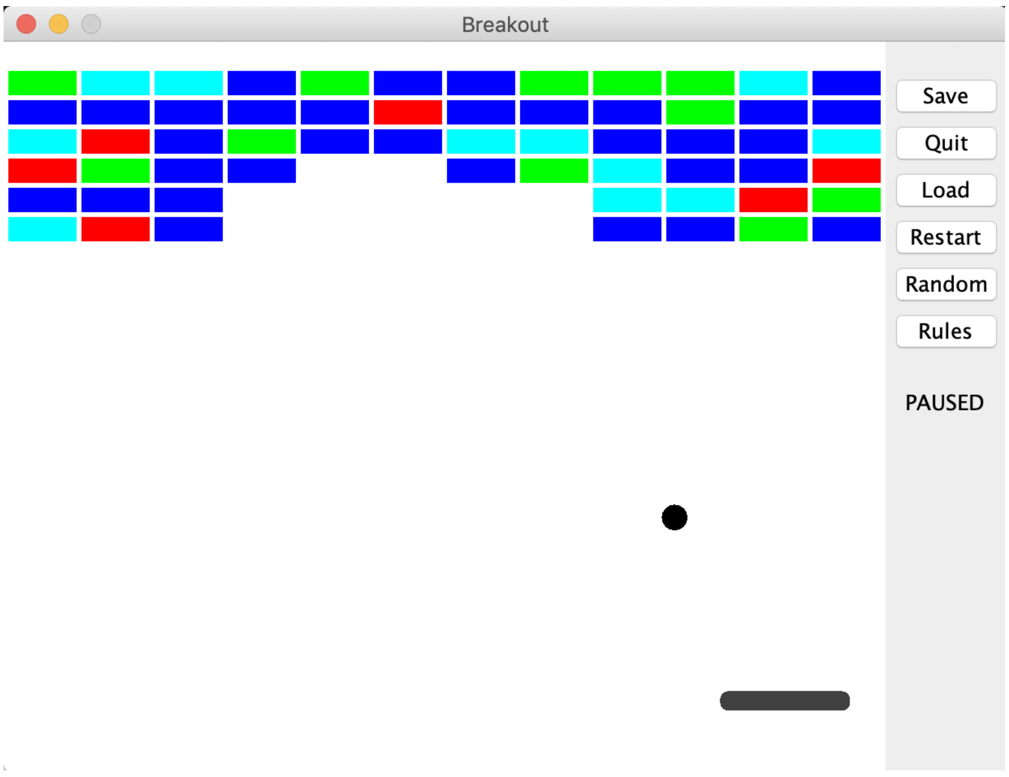
- **BrickType.java**: This enum lists the different possible types for bricks, and also stores the distribution for how those bricks are randomly assigned when the `Wall` is constructed.
- **GameState.java**: This enum lists the different possible states of the game.

And that's it! It is kind of a lot of code, but there's lot of library imports, general Java boilerplate, and text outputs, so the actual code running the game won't be too verbose.

Images

I've also including two pitures of what the game actually looks like in case you're curious!





Files

Below I've attached all of the `.java` files (also included in zip file). Feel free to search by name if you're looking for one in particular.

Ball.java

```java
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.Serializable;
import java.util.Random;

@SuppressWarnings("serial")
public class Ball implements Serializable {
    static final int BALL_RADIUS = 16;
    static final int STARTING_VELOCITY = 4;
    private int vx;
    private int vy;
    private int x;
    private int y;

    // Constructors
    public Ball() {
        this.x = 0;
        this.y = 0;

        Random random = new Random();
        int xSign = random.nextBoolean() ? 1 : -1;
        this.vx = xSign * STARTING_VELOCITY;
        this.vy = STARTING_VELOCITY;
    }

    // Getters
    public int getVX() {
        return this.vx;
    }

    public int getVY() {
        return this.vy;
    }

    public void draw(Graphics gc) {
        Graphics2D gc2d = (Graphics2D)gc;
        gc2d.setColor(Color.black);
        gc2d.fillOval(x, y, BALL_RADIUS, BALL_RADIUS);
    }
    // Method called during the game loop -- moves the ball forward
    // by its velocity and then checks any of its collisions with
    // either the paddle, a brick, or the edge of the screen
    public void move(Game game, Paddle paddle, Wall wall) {
        double screenRadius = 0.5 * Game.SCREEN_WIDTH;

        int paddleRadius = (int) (Paddle.PADDLE_WIDTH * 0.5);
        int paddleLeft = paddle.getX() - paddleRadius;
        int paddleRight = paddle.getX() + paddleRadius;

        int halfBallRadius = (int) (Ball.BALL_RADIUS * 0.5);
        int paddleHeight = (int) (Paddle.PADDLE_HEIGHT * 0.5);
        int heightRadius = (int) (0.5 * Game.SCREEN_HEIGHT);
        int paddleY = Paddle.PADDLE_Y - heightRadius;
        int paddleTop = paddleY - paddleHeight - (int)(Ball.BALL_RADIUS *
0.5);
        int paddleBottom = paddleY + paddleHeight;

        if (this.x < -screenRadius || screenRadius - Ball.BALL_RADIUS <
this.x) {
            // Collision with side of screen
            this.vx = -this.vx;
            this.x += this.vx;
        } else if (paddleLeft <= this.x + halfBallRadius
                && this.x - halfBallRadius <= paddleRight
                && paddleTop <= this.y && this.y <= paddleBottom) {
            // Collision with the paddle
            int distance = Math.abs(this.x - paddle.getX());
            if (distance > (int)(paddleRadius * 0.3)) {
                if (this.vx < 0) {
                    this.vx -= 1;
                    this.vx = Math.max(this.vx, -6);
                } else {
                    this.vx += 1;
                    this.vx = Math.min(this.vx, 6);
                }
            } else {
                if (this.vx < 0) {
                    this.vx += 1;
                    this.vx = Math.max(this.vx, -6);
                } else {
                    this.vx -= 1;
                    this.vx = Math.min(this.vx, 6);
                }
            }

            this.vy = -this.vy;
            this.y += this.vy;
        } else if (this.y < -heightRadius) {
            // Collision with top of screen
            this.vy = -this.vy;
            this.y += this.vy;
        } else if (this.y > heightRadius) {
            // Collision with bottom of screen (game over!)
```

```java
                game.setGameState(GameState.LOST);
            } else {
                boolean wallCollision = wall.collide(this.x, this.y, this);

                // Collision with the wall (any of the bricks)
                if (wallCollision) {
                    if (wall.checkWin()) {
                        game.setGameState(GameState.WON);
                    }
                    this.vy = -this.vy;
                    this.y += this.vy;
                } else {
                    this.x = this.x + this.vx;
                    this.y = this.y + this.vy;
                }
            }
        }

        // Method to be called when the ball bounces off of a brick with
        // BrickType of STICK
        public void stick() {
            if (this.vx < 0) {
                this.vx -= 1;
            } else {
                this.vx += 1;
            }

            if (this.vy < 0) {
                this.vy -= 1;
            } else {
                this.vy += 1;
            }
        }

        // Serializer methods
        public void writeSerializedText(BufferedWriter writer) throws
    IOException {
            writer.write(x + " " + y + " " + vx + " " + vy);
            writer.newLine();
        }

        public void readSerializedText(BufferedReader reader) throws
    IOException {
            String[] values = reader.readLine().split(" ");
            x = Integer.parseInt(values[0]);
            y = Integer.parseInt(values[1]);
            vx = Integer.parseInt(values[2]);
            vy = Integer.parseInt(values[3]);
        }
    }
```

Paddle.java

```java
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;

public class Paddle {
    static final int PADDLE_Y = 400;
    static final int PADDLE_WIDTH = 80;
    static final int PADDLE_HEIGHT = 12;
    static final int PADDLE_ARC_WIDTH = 10;
    static final int PADDLE_ARC_HEIGHT = 10;
    static final int PADDLE_VELOCITY = 9;
    static final int RIGHT_VELOCITY = PADDLE_VELOCITY;
    static final int LEFT_VELOCITY = -PADDLE_VELOCITY;
    private int x;
    private int v;

    // Constructors
    public Paddle() {
        x = 0;
        v = 0;
    }

    // Getters
    public int getX() {
        return x;
    }

    // Setters
    public void setX(int x) {
        this.x = x;
    }

    public void setV(int v) {
        this.v = v;
    }

    public void draw(Graphics gc) {
        Graphics2D gc2d = (Graphics2D)gc;
        gc2d.setColor(Color.DARK_GRAY);

        gc2d.fillRoundRect(x, 0, PADDLE_WIDTH, PADDLE_HEIGHT,
                           PADDLE_ARC_WIDTH, PADDLE_ARC_HEIGHT);
    }

    // Move paddle based on user input
    public void move() {
        int newX = this.x + this.v;
        double screenRadius = 0.5 * Game.SCREEN_WIDTH;
        if (newX < PADDLE_WIDTH * 0.5 - screenRadius) {
            this.x = (int) (PADDLE_WIDTH * 0.5 - screenRadius);
        } else if (newX > screenRadius - PADDLE_WIDTH * 0.5) {
            this.x = (int) (screenRadius - PADDLE_WIDTH * 0.5);
        } else {
            this.x = newX;
        }
    }

    // Serializer methods
    public void writeSerializedText(BufferedWriter writer) throws
IOException {
        writer.write(x + "");
        writer.newLine();
    }

    public void readSerializedText(BufferedReader reader) throws
IOException {
        x = Integer.parseInt(reader.readLine());
    }
}
```

Wall.java

```
import java.awt.Graphics;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.util.LinkedList;
import java.util.List;
import java.util.Random;

public class Wall {
    public static final int WALL_Y = 15;
    int rows;
    int columns;
    Brick[][] bricks;
    List<Brick> brokenBricks;

    // Constructors
    public Wall(int rows, int columns, boolean random) {
        this.rows = rows;
        this.columns = columns;
        this.bricks = new Brick[rows][columns];
        this.brokenBricks = new LinkedList<>();

        for (int r = 0; r < rows; r++) {
            for (int c = 0; c < columns; c++) {
                if (!random) {
                    bricks[r][c] = new Brick(r, c);
                } else {
                    bricks[r][c] = new Brick(r, c,
BrickType.getRandomType());
                }
            }
        }
    }

    public void draw(Graphics gc) {
        for (int r = 0; r < rows; r++) {
            for (int c = 0; c < columns; c++) {
                Brick current = bricks[r][c];
                if (current != null) {
                    current.draw(gc);
                }
            }
        }
    }

    // Method to be called when we are breaking the brick at row = r and
    // column = c. We set the BrickType to be BROKEN (which means it isn't
    // drawn and is not considered in collisions)
    public void hit(int r, int c, Ball ball) {;
        if (bricks[r][c].getType() != BrickType.BROKEN && bricks[r]
[c].hit()) {
            BrickType oldType = bricks[r][c].getType();
            brokenBricks.add(bricks[r][c]);
            bricks[r][c].setType(BrickType.BROKEN);
            // Explosive bricks also break the bricks adjacent to them
when
            // they are broken -- we acomplish this using recursion on
this
            // method which each adjacent brick
            if (oldType == BrickType.EXPLOSIVE) {
                int rStart = Math.max(r - 1, 0);
                int rEnd = Math.min(r + 1, rows - 1);

                int cStart = Math.max(c - 1, 0);
                int cEnd = Math.min(c + 1, columns - 1);

                for (int i = rStart; i <= rEnd; i++) {
                    for (int j = cStart; j <= cEnd; j++) {
                        if (!(i == r && j == c)) {
                            hit(i, j, ball);
                        }
                    }
                }
            } else if (oldType == BrickType.RELOAD &&
                    !(brokenBricks == null || brokenBricks.size() <= 1)) {
                // Reload bricks reload a previous brick when broken
(setting
                // its state from BROKEN to whatever its previous state
was)

                // We want to pick the brick before the actual RELOAD
                // brick that we just inserted
                Brick reload = brokenBricks.get(brokenBricks.size() - 2);

                if (unhit(reload.getR(), reload.getC())) {
                    brokenBricks.remove(brokenBricks.size() - 2);
                }
            } else if (oldType == BrickType.STICK) {
                // Stick bricks send the balls back in the direction that
                // they were coming from making it difficult to predict
                ball.stick();
            }



        }
    }

    // Method to "unbreak" a brick by setting its BrickType from BROKEN to
    // its original type
    public boolean unhit(int r, int c) {
        if (bricks[r][c].hit()) {
            bricks[r][c].setType(bricks[r][c].getOriginalType());
            return true;
        }

        return false;
    }

    // Check to see if the ball is colliding with any of the bricks
    public boolean collide(int x, int y, Ball ball) {
        for (int r = 0; r < rows; r++) {
            for (int c = 0; c < columns; c++) {
                Brick current = bricks[r][c];

                if (current.getType() == BrickType.BROKEN) {
                    continue;
                }

                int xBrick = current.getX() - (int) (0.5 *
Game.SCREEN_WIDTH);
                int yBrick = WALL_Y + current.getY()
                        - (int) (0.5 * Game.SCREEN_HEIGHT);

                int brickLeft = xBrick - Brick.BRICK_WIDTH +
Brick.BRICK_PADDING;
                int brickRight = xBrick + Brick.BRICK_WIDTH -
Brick.BRICK_PADDING;
                int brickTop = yBrick - Brick.BRICK_HEIGHT +
Brick.BRICK_PADDING;
                int brickBottom = yBrick + Brick.BRICK_HEIGHT -
Brick.BRICK_PADDING;

                if (brickLeft <= x && x + Ball.BALL_RADIUS <= brickRight
                    && brickTop <= y && y <= brickBottom) {
                    hit(r, c, ball);
                    return true;
                }
            }
        }

        return false;
    }

    // Serializer methods
    public void writeSerializedText(BufferedWriter writer) throws
IOException {
        for (int r = 0; r < rows; r++) {
            for (int c = 0; c < columns; c++) {
                Brick current = bricks[r][c];
                if (current != null) {
                    writer.write(current.getType().name());
                    writer.write(" ");
                }
            }
            writer.newLine();
        }
    }

    public void readSerializedText(BufferedReader reader) throws
IOException {
        for (int r = 0; r < rows; r++) {
            String[] strings = reader.readLine().split(" ");
            for (int c = 0; c < columns; c++) {
                BrickType type = BrickType.valueOf(strings[c]);
                bricks[r][c].setType(type);
            }
        }
    }

    public boolean checkWin() {
        for (int r = 0; r < rows; r++) {
            for (int c = 0; c < columns; c++) {
                Brick current = bricks[r][c];
                if (current.getType() != BrickType.BROKEN) {
                    return false;
                }
            }
        }

        return true;
    }
}
```

Brick.java

```java
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;

public class Brick {
    static final int BRICK_WIDTH = 45;
    static final int BRICK_HEIGHT = 18;
    static final int BRICK_PADDING = 3;

    private int r;
    private int c;
    private int x;
    private int y;
    private int health;
    private BrickType type;
    private BrickType originalType;

    // Constructors
    public Brick(int r, int c) {
        this.r = r;
        this.c = c;
        this.x = BRICK_WIDTH * c;
        this.y = BRICK_HEIGHT * r;
        this.health = 1;
        this.type = BrickType.NORMAL;
        this.originalType = this.type;
    }

    public Brick(int r, int c, BrickType type) {
        this.r = r;
        this.c = c;
        this.x = BRICK_WIDTH * c;
        this.y = BRICK_HEIGHT * r;
        this.health = 1;
        this.type = type;
        this.originalType = this.type;
    }

    // Getters
    public int getX() {
        return this.x;
    }

    public int getY() {
        return this.y;
    }

    public int getR() {
        return this.r;
    }

    public int getC() {
        return this.c;
    }

    public BrickType getType() {
        return this.type;
    }

    public BrickType getOriginalType() {
        return this.originalType;
    }

    // Setters
    public void setType(BrickType type) {
        this.type = type;
    }

    // Returns true if this hit breaks the brick
    public boolean hit() {
        health--;
        return health <= 0;
    }

    // Returns true if this hit brings back the brick
    public boolean unhit() {
        health++;
        return health > 0;
    }

    public Color getColor() {
        switch (type) {
            case NORMAL: return Color.BLUE;
            case EXPLOSIVE: return Color.RED;
            case STICK: return Color.CYAN;
            case RELOAD: return Color.GREEN;
        }

        return Color.BLUE;
    }

    public void draw(Graphics gc) {
        if (type == BrickType.BROKEN) {
            return;
        }

        Graphics2D gc2d = (Graphics2D)gc;
        gc2d.setColor(getColor());
        gc2d.fillRect(this.x + BRICK_PADDING, this.y + BRICK_PADDING,
                      BRICK_WIDTH - BRICK_PADDING,
                      BRICK_HEIGHT - BRICK_PADDING);
    }
}
```

Game.java

```java
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.sql.Timestamp;
import java.util.Timer;
import java.util.TimerTask;

import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;
import javax.swing.filechooser.FileNameExtensionFilter;

public class Game implements Runnable {

    JFrame frame;
    Wall wall;
    Paddle paddle;
    Ball ball;
    GameCanvas game;
    GameState gameState;
    Menu menu;
    static final int WALL_ROWS = 6;
    static final int WALL_COLUMNS = 12;
    static final int SCREEN_WIDTH = Brick.BRICK_WIDTH * WALL_COLUMNS
                                + Brick.BRICK_PADDING;
    static final int SCREEN_HEIGHT = 450;
    static final int GAME_STEP_INTERVAL = 20;

    @Override
    public void run() {
        frame = new JFrame("Breakout");
        frame.setLayout(new BorderLayout());
        game = this.new GameCanvas();
        game.setFocusable(true);
        wall = new Wall(WALL_ROWS, WALL_COLUMNS, false);
        paddle = new Paddle();
        ball = new Ball();

        menu = new Menu();
        menu.setVisible(true);
        setGameState(GameState.PAUSED);

        Key keyListener = new Key();
        game.addKeyListener(keyListener);

        frame.add(game, BorderLayout.CENTER);
        frame.add(menu, BorderLayout.LINE_END);
        frame.pack();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
        frame.setResizable(false);

        Timer timer = new Timer();
        timer.schedule(new TimerTask() {
            @Override
            public void run() {
                tick();
            }
        }, 0, GAME_STEP_INTERVAL);
    }

    @SuppressWarnings("serial")
    public class GameCanvas extends JPanel {
        // Draws the actual game board by creating a graphics context
        // and passing that graphics context to the draw method of each
        // individual object on the screen (translating the graphics
context
        // allows us to be able to do relative positioning when printing
        // the objects)
        @Override
        public void paintComponent(Graphics gc) {
            super.paintComponent(gc);

            gc.translate(0, Wall.WALL_Y);
            wall.draw(gc);

            int x = (int) (SCREEN_WIDTH * 0.5 - Paddle.PADDLE_WIDTH *
0.5);
            int y = (int) (Paddle.PADDLE_Y - Wall.WALL_Y);
            gc.translate(x, y);
            paddle.draw(gc);

            x = (int) (Paddle.PADDLE_WIDTH * 0.5);
            y = (int) (0.5 * SCREEN_HEIGHT - Paddle.PADDLE_Y);
            gc.translate(x, y);
            ball.draw(gc);
        }
```

```java
        @Override
        public Dimension getPreferredSize() {
            return new Dimension(SCREEN_WIDTH, SCREEN_HEIGHT);
        }

        public GameCanvas() {
            super();
            setBackground(Color.WHITE);
        }

        public void setGameState(GameState paused) {
            // TODO Auto-generated method stub

        }
    }

    @SuppressWarnings("serial")
    public class Menu extends JPanel {
        static final int MENU_WIDTH = 75;

        private JButton save;
        private JButton quit;
        private JButton load;
        private JButton restart;
        private JButton randomize;
        private JButton rules;
        private JLabel stateLabel;

        @Override
        public void paintComponent(Graphics gc) {
            super.paintComponent(gc);
        }

        @Override
        public Dimension getPreferredSize() {
            return new Dimension(MENU_WIDTH, SCREEN_HEIGHT);
        }

        // Pause menu for the game loop -- includes Save and Load buttons
        // which save to and load from serialized files
        public Menu() {
            super();
            setBackground(new Color(238, 238, 238));
            setLayout(new BoxLayout(this, BoxLayout.PAGE_AXIS));

            this.save = new JButton("Save");
            this.quit = new JButton("Quit");
            this.load = new JButton("Load");
            this.restart = new JButton("Restart");
            this.randomize = new JButton("Random");
            this.rules = new JButton("Rules");
            this.stateLabel = new JLabel();
            add(new Box.Filler(new Dimension(0, 20),
                               new Dimension(0, 20),
                               new Dimension(0, 20)));
            add(save);
            add(quit);
            add(load);
            add(restart);
            add(randomize);
            add(rules);
            add(new Box.Filler(new Dimension(0, 20),
                        new Dimension(0, 20),
                        new Dimension(0, 20)));
            add(stateLabel);

            // Callback method for serializing game state when Save is
clicked
            save.addActionListener(new ActionListener() {
                @Override
                public void actionPerformed(ActionEvent e) {
                    JFileChooser chooser = new JFileChooser();

chooser.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
                    chooser.setDialogTitle("Select a directory to save
your game file");
                    chooser.showOpenDialog(null);

                    String filename = "Breakout-"
                                            + (new
Timestamp(System.currentTimeMillis()))
                                            + ".break";
                    File saveLocation = chooser.getSelectedFile();
                    try {
                        FileWriter fw = new
FileWriter(saveLocation.getAbsolutePath()
                                                                    + "/" +
filename, true);
                        BufferedWriter writer = new BufferedWriter(fw);
                        writeSerializedText(writer);
                        writer.flush();
                        writer.close();
                    } catch (FileNotFoundException exception) {
                        throw new IllegalArgumentException();
                    } catch (IOException exception) {
                        throw new IllegalArgumentException();
                    }

                    game.requestFocus();
                    return;
                }
            });

            // Callback method for quitting the game when Quit is clicked
            quit.addActionListener(new ActionListener() {
                @Override
                public void actionPerformed(ActionEvent e) {
                    System.exit(0);
                }
            });

            // Callback method for loading the game state from a file
            // when Load is clicked
            load.addActionListener(new ActionListener() {
                @Override
                public void actionPerformed(ActionEvent e) {
                    JFileChooser chooser = new JFileChooser();
                    FileNameExtensionFilter filter =
                            new FileNameExtensionFilter("Breakout Save
(.break)", "break");
                    chooser.setDialogTitle("Select a saved game file");
                    chooser.setFileFilter(filter);

                    chooser.showOpenDialog(null);
                    File saveLocation = chooser.getSelectedFile();

                    if (saveLocation == null ||
!saveLocation.getPath().contains(".break")) {
                        game.requestFocus();
                        return;
                    }

                    try {
                        FileReader fr = new
FileReader(saveLocation.getAbsolutePath());
                        BufferedReader writer = new BufferedReader(fr);
                        readSerializedText(writer);
                    } catch (FileNotFoundException exception) {
                        throw new IllegalArgumentException();
                    }

                    game.requestFocus();
                    return;
                }
            });

            // Callback method for restarting the game
            restart.addActionListener(new ActionListener() {
                @Override
                public void actionPerformed(ActionEvent e) {
                    wall = new Wall(WALL_ROWS, WALL_COLUMNS, false);
                    ball = new Ball();

                    setGameState(GameState.PAUSED);
                    game.requestFocus();
                }
            });

            // Callback method for randomizing the block types (useful if
you
```

```
            // don't like your current arrangment)
            randomize.addActionListener(new ActionListener() {
                @Override
                public void actionPerformed(ActionEvent e) {
                    wall = new Wall(WALL_ROWS, WALL_COLUMNS, true);
                    ball = new Ball();

                    setGameState(GameState.PAUSED);
                    game.requestFocus();
                }
            });

            // Callback method for printing out the rules to the screen
            // in HTML formatted text
            rules.addActionListener(new ActionListener() {
                @Override
                public void actionPerformed(ActionEvent e) {
                    String normal = "BLUE bricks are NORMAL and won't do"
                            + " anything when you break them.";
                    String stick = "AQUA-MARINE bricks are STICK and will
speed"
                            + " up the ball when they are hit.";
                    String explosive = "RED bricks are EXPLOSIVE and will
break"
                            + " surrounding bricks when hit.";
                    String reload = "GREEN bricks are RELOAD and will
respawn"
                            + " the most recently broken brick when"
                            + " they are broken";
                    String explaination = "The objective is to break all
of "
                            + "the bricks without allowing the ball to
fall "
                            + "below your paddle.";
                    String paddle1 = "The ball will bounce according to
how "
                            + "it hits your paddle.";
                    String paddle2 = "Closer to the center is "
                            + "more vertical. Closer to the edge is more "
                            + "horizontal";
                    String wall = "The wall starts off as being all NORMAL
"
                            + "bricks. Hit 'random' to randomize the
bricks.";
                    String save = "Hit save/load to save your current game
or"
                            + " load in an old game.";
                    String space = "Most importantly: HIT SPACE TO
PAUSE/UNPAUSE"
                            + " THE GAME!";
                    String goodluck = "Good luck!";

                    String text = "<html>"
                            + "<p>" + normal + "</p>"
                            + "<p>" + stick + "</p>"
                            + "<p>" + explosive + "</p>"
                            + "<p>" + reload + "</p>"
                            + "<p></p>"
                            + "<p>" + explaination + "</p>"
                            + "<p></p>"
                            + "<p>" + paddle1 + "</p>"
                            + "<p>" + paddle2 + "</p>"
                            + "<p></p>"
                            + "<p>" + wall + "</p>"
                            + "<p></p>"
                            + "<p>" + save + "</p>"
                            + "<p></p>"
                            + "<p>" + space + "</p>"
                            + "<p></p>"
                            + "<p></p>"
                            + "<p>" + goodluck + "</p>"
                            + "</html>";

                    JOptionPane.showMessageDialog(null, text);
                    setGameState(GameState.PAUSED);
                    game.requestFocus();
                }
            });
        }

        // Get the stateLabel so user can see current state of the game
        // (PLAYING, PAUSED, LOST, WON).
        public void setStateText(GameState state) {
            stateLabel.setText("   " + state.toString());
        }
    }

    // KeyAdapter allows us to parse input from user and change the
velocity
    // of the paddle (also if press the space key it will pause the game)
    public class Key extends KeyAdapter {

        @Override
        public void keyPressed(KeyEvent e) {
            if (e.getKeyCode() == KeyEvent.VK_LEFT) {
                paddle.setV(Paddle.LEFT_VELOCITY);
            } else if (e.getKeyCode() == KeyEvent.VK_RIGHT) {
                paddle.setV(Paddle.RIGHT_VELOCITY);
            } else if (e.getKeyCode() == KeyEvent.VK_SPACE) {
                if (gameState == GameState.PLAYING) {
                    setGameState(GameState.PAUSED);
                } else if (gameState == GameState.PAUSED) {
                    setGameState(GameState.PLAYING);
                }
            }
        }

        @Override
        public void keyReleased(KeyEvent e) {
            if (e.getKeyCode() == KeyEvent.VK_LEFT
                    || e.getKeyCode() == KeyEvent.VK_RIGHT) {
                paddle.setV(0);
            }
        }
    }

    // This is the method run at every iteration of the game loop
    // essentially checking the game state and performing the appropriate
    // operations (most principle being repainting)
    protected void tick() {
        if (gameState == GameState.PLAYING) {
            paddle.move();
            // Collisions between paddle and ball checked in
            // ball class -- this is where the change in velocity
            // happens, so it felt like the most natural place
            ball.move(this, paddle, wall);
            game.repaint();
        } else if (gameState == GameState.PAUSED) {
            game.repaint();
            menu.repaint();
        } else if (gameState == GameState.LOST) {
            game.repaint();
            menu.repaint();
        } else if (gameState == GameState.WON) {
            game.repaint();
            menu.repaint();
        }
    }


    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Game());
    }

    public void setGameState(GameState gameState) {
        menu.setStateText(gameState);
        this.gameState = gameState;
    }

    // Serializer methods for the actual Game (this just calls serializer
    // methods that are defined in other classes)
    public void writeSerializedText(BufferedWriter writer) {
        try {
            wall.writeSerializedText(writer);
            writer.newLine();
            paddle.writeSerializedText(writer);
            writer.newLine();
            ball.writeSerializedText(writer);
        } catch (IOException e) {
            return;
        }
    }
    public void readSerializedText(BufferedReader reader) {
        try {
            wall.readSerializedText(reader);
            reader.readLine();
            paddle.readSerializedText(reader);
            reader.readLine();
            ball.readSerializedText(reader);
        } catch (IOException e) {
            return;
        }
    }

}
```

BrickType.java

```
public enum BrickType {
    NORMAL, // Normal brick in breakout
    EXPLOSIVE, // Also breaks adjacent bricks when broken
    STICK, // Sends the ball back in the direction it was coming
    RELOAD, // Reloads the next most recently broken brick when broken
    BROKEN; // Is not printed to the screen and will not be considered in
collisions

    public static BrickType getRandomType() {
        double random = Math.random();
        BrickType returnType;

        // Custom probability distribution for BrickTypes
        // NORMAL: 55%
        // EXPLOSIVE: 15%
        // STICK: 15%
        // RELOAD: 15%
        if (random <= 0.55) {
            returnType = NORMAL;
        } else if (random <= 0.7) {
            returnType = EXPLOSIVE;
        } else if (random <= 0.85) {
            returnType = STICK;
        } else {
            returnType = RELOAD;
        }

        return returnType;
    }
}
```

GameState.java

```java
public enum GameState {
    PLAYING,
    PAUSED,
    LOST,
    WON
}
```