```java
package pacman;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;
/**
 * This is the App class where your Pacman game will start.
 * The main method of this application calls the start method. You
 * will need to fill in the start method to instantiate your game.
 *
 * Class comments here...
 *
 */
public class App extends Application {
    private BorderPane root;
    @Override
    public void start(Stage stage) {
        // Create top-level object, set up the scene, and show the stage here.
        PaneOrganizer organizer = new PaneOrganizer();
        root = new BorderPane();
        Scene scene = new Scene(organizer.getRoot(), 575, 634);
        stage.setTitle("Pacman");
        stage.setScene(scene);
        stage.show();
    }
    /**
     * Here is the mainline! No need to change this.
     */
    public static void main(String[] argv) {
        // launch is a method inherited from Application
        launch(argv);
    }
}
package pacman;
/**
 * This class is an immutable representation of some coordinate within the
 * Pacman board world. As the board consists of square blocks arranged in a
 * 23x23 grid, all elements/blocks in the game must exist within this coordinate
 * space. *However*, when creating targets in Chase mode, your target may be out
 * of the bounds of the board -- that is okay for this scenario only. Therefore,
 * you can explicitly override the bounds-checking functionality by setting
 * isTarget to true in the constructor's third parameter. You should *only*
 * set this parameter to true if, because of the game logic, you specifically
 * want to allow out of bounds squares to be created (essentially, only when
 * creating targets).
 *
 * "Immutable" simply means that once you create the object, you can't change
 * anything about it - notice that while we have getter methods for the row and
 * column, there aren't any setters!
 *
 * There are two purposes to using this class instead of a vanilla
 * java.awt.Point or javafx.geometry.Point2D. First, we are able to bound the
 * ranges that the row and columns for this coordinate are able to take on,
 * thereby reducing bugs which would otherwise lead to ArrayIndexOutOfBounds
 * exceptions being thrown in mysterious ways. Second, by naming the dimensions
```

```
 * "row" and "column" rather than X and Y, the hope is to make clearer the
 * "true" location of each coordinate location.
 */
public class BoardCoordinate {
    private final int row;
    private final int column;
    private static final int ROW_MAX = 22;
    private static final int COL_MAX = 22;
    /**
     * The constructor. it takes in a row and a column whose location this
     * instance will model, and a boolean of whether the square is a target
     * square (see header comments for more on this).
     */
    public BoardCoordinate(int row, int column, boolean isTarget) {
        if (!isTarget) {
            this.checkValidity(row, column);
        }
        this.row = row;
        this.column = column;
    }
    /**
     * Returns the row index that this BoardCoordinate represents.
     */
    public int getRow() {
        return this.row;
    }
    /**
     * Returns the column index that this BoardCoordinate represents.
     */
    public int getColumn() {
        return this.column;
    }
    /**
     * Checks that the row and index passed into this class' constructor are
     * bounded by 0 and the ROW_MAX for the row and the COL_MAX for the column,
     * respectively.
     *
     * NOTE: You've seen exceptions like ArrayIndexOutOfBounds exceptions being
     * thrown before, but you haven't seen what code that generates these
     * Exceptions looks like - Here, we throw an IllegalArgumentException if the
     * row and column parameters are invalid! Try instantiating an instance of
     * this class with invalid coordinates, e.g. (-4, 50) and see what happens
     * when you run the code! Runtime Exceptions like IllegalArgumentExceptions
     * and ArrayIndexOutOfBounds Exceptions usually indicate that something is
     * wrong with the code, and needs to be fixed.
     */
    private void checkValidity(int row, int column) {
        if (row < 0 || column < 0) {
            throw new IllegalArgumentException("Board Coordinates must not be
negative: " + " Given row = " + row + " col = " + column);
        } else if (row > ROW_MAX || column > COL_MAX) {
            throw new IllegalArgumentException("Board Coordinates must not exceed
board dimensions: " + " Given row = " + row + " col = " + column);
        }
    }
}
package pacman;
```

```java
/**
 * This is the interface Collidable, created to unify the ghosts, the dots, and
 * the energizers using their similarities.
 */
public interface Collidable {
    void removeDotFromPane();
    void removeEnergizerFromPane();
    boolean isEnergizer();
    boolean isDot();
    boolean isGhost();
}
package pacman;
/**
 * Constants class.
 */
public class Constants {
    public static final int SQUARE = 25;
}
package pacman;
/**
 * Direction enum class.
 */
public enum Direction {
    UP,
    DOWN,
    LEFT,
    RIGHT;
    /**
     * Method defines what opposite for each of the directions.
     */
    public Direction opposite() {
        switch (this) {
            case UP:
                return DOWN;
            case DOWN:
                return UP;
            case LEFT:
                return RIGHT;
            default:
                return LEFT;
        }
    }
}
package pacman;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
/**
 * Dot class, implements the interface Collidable.
 */
public class Dot implements Collidable {
    private Circle dot;
    private Pane gamePane;
    private int row;
    private int col;
    private ScoreController controller;
    /**
```

```
168         * Constructor of the Dot class.
169         */
170        public Dot(int row, int col, Pane gamePane, ScoreController controller) {
171            super();
172            this.controller = controller;
173            this.gamePane = gamePane;
174            this.row = row;
175            this.col = col;
176            this.dot = new Circle((col * Constants.SQUARE) + Constants.SQUARE / 2, (row *
    Constants.SQUARE) + Constants.SQUARE / 2, 2);
177            this.dot.setFill(Color.WHITESMOKE);
178            this.gamePane.getChildren().add(this.dot);
179        }
180        /**
181         * Interface methods.
182         */
183        /**
184         * Method allows to remove the Circle Dot from the pane, used when pacman collides
    with a dot
185         * in the game class.
186         */
187        @Override
188        public void removeDotFromPane() {
189            this.gamePane.getChildren().remove(this.dot);
190        }
191        @Override
192        public void removeEnergizerFromPane() {}
193        @Override
194        public boolean isEnergizer() {
195            return false;
196        }
197        /**
198         * Returns true to boolean isDot.
199         */
200        @Override
201        public boolean isDot() {
202            return true;
203        }
204        @Override
205        public boolean isGhost() {
206            return false;
207        }
208    }
209    package pacman;
210    import javafx.scene.layout.Pane;
211    import javafx.scene.paint.Color;
212    import javafx.scene.shape.Circle;
213    /**
214     * Energizer class. Implements interface Collidable.
215     */
216    public class Energizer implements Collidable {
217        private Circle energizer;
218        private Pane gamePane;
219        private int row;
220        private int col;
221        /**
222         * Energizer class constructor.
```

```
     */
    public Energizer(int row, int col, Pane gamePane) {
        this.gamePane = gamePane;
        this.row = row;
        this.col = col;
        this.energizer = new Circle((col * Constants.SQUARE) + Constants.SQUARE / 2,
(row * Constants.SQUARE) + Constants.SQUARE / 2, 6);
        this.energizer.setFill(Color.WHITESMOKE);
        this.gamePane.getChildren().add(this.energizer);
    }
    /**
     * Interface methods.
     */
    @Override
    public void removeDotFromPane() {}
    /**
     * Method allows to remove the Circle Energizer from the pane, used when pacman
collides with
     * an energizer in the game class.
     */
    @Override
    public void removeEnergizerFromPane() {
        this.gamePane.getChildren().remove(this.energizer);
    }
    /**
     * Returns true to boolean isEnergizer.
     */
    @Override
    public boolean isEnergizer() {
        return true;
    }
    @Override
    public boolean isDot() {
        return false;
    }
    @Override
    public boolean isGhost() {
        return false;
    }
}
package pacman;
import javafx.animation.Animation;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.scene.control.Label;
import javafx.scene.layout.Pane;
import cs15.fnl.pacmanSupport.CS15SupportMap;
import cs15.fnl.pacmanSupport.CS15SquareType;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.util.Duration;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;
/**
 * Game class.
 */
```

```
278 public class Game {
279     private MazeSquares[][] boardArray;
280     private MazeSquares squares;
281     private Pane gamePane;
282     private Timeline generalTimeline;
283     private Pacman pacman;
284     private Dot dot;
285     private Energizer energizer;
286     private Ghost ghost;
287     private Ghost ghostOne;
288     private Ghost ghostTwo;
289     private Ghost ghostThree;
290     private ScoreController controller;
291     private int gt;
292     private int scatter;
293     private int chase;
294     private int comeFantasmas;
295     private int hearts;
296     private Queue < Ghost > ghosts;
297     private Modes mode;
298     /**
299      * Constructor of the game class.
300      */
301     public Game(Pane gamePane, MazeSquares[][] boardArray, ScoreController controller,
    MazeSquares squares) {
302         this.gt = 10;
303         this.scatter = 14;
304         this.chase = 20;
305         this.comeFantasmas = 16;
306         this.hearts = 0;
307         this.mode = Modes.CHASE;
308         this.controller = controller;
309         this.squares = squares;
310         this.gamePane = gamePane;
311         this.boardArray = boardArray;
312         this.pacman = new Pacman(gamePane, boardArray);
313         this.createBoard();
314         this.setUpGeneralTimeline();
315         this.ghost.sendToFront();
316         this.ghostOne.sendToFront();
317         this.ghostTwo.sendToFront();
318         this.ghostThree.sendToFront();
319         this.pacman.setBoardArray(this.boardArray);
320         this.pacman.toFront();
321     }
322     /**
323      * Method that creates the board of MazeSquares using the Map to create each
    element
324      * on its correct position.
325      */
326     public void createBoard() {
327         CS15SquareType[][] board = CS15SupportMap.getSupportMap();
328         for (int row = 0; row < 23; row++) {
329             for (int col = 0; col < 23; col++) {
330                 boardArray[row][col] = new MazeSquares(row, col, gamePane);
331             }
332         }
```

```
333         for (int row = 0; row < 23; row++) {
334             for (int col = 0; col < 23; col++) {
335                 switch (board[row][col]) {
336                     case DOT:
337                         this.dot = new Dot(row, col, gamePane, controller);
338                         this.boardArray[row][col].getArrayList().add(dot);
339                         break;
340                     case WALL:
341                         this.boardArray[row][col].newColor(Color.BLUE);
342                         break;
343                     case ENERGIZER:
344                         this.energizer = new Energizer(row, col, gamePane);
345                         this.boardArray[row][col].getArrayList().add(this.energizer);
346                         break;
347                     case GHOST_START_LOCATION:
348                         this.ghostOne = new Ghost(row - 2, col, gamePane, boardArray,
    pacman, controller);
349                         this.ghostOne.setColor(Color.RED);
350                         this.boardArray[row - 2][col].getArrayList().add(ghostOne);
351                         this.ghost = new Ghost(row, col, gamePane, boardArray, pacman,
    controller);
352                         this.ghost.setColor(Color.GREEN);
353                         this.boardArray[row][col].getArrayList().add(ghost);
354                         this.ghostTwo = new Ghost(row, col - 1, gamePane, boardArray,
    pacman, controller);
355                         this.ghostTwo.setColor(Color.SKYBLUE);
356                         this.boardArray[row][col - 1].getArrayList().add(ghostTwo);
357                         this.ghostThree = new Ghost(row, col + 1, gamePane,
    boardArray, pacman, controller);
358                         this.ghostThree.setColor(Color.YELLOW);
359                         this.boardArray[row][col + 1].getArrayList().add(ghostThree);
360                         this.ghosts = new LinkedList < > ();
361                         this.ghosts.add(this.ghost);
362                         this.ghosts.add(this.ghostTwo);
363                         this.ghosts.add(this.ghostThree);
364                         break;
365                     default:
366                         break;
367                 }
368             }
369         }
370     }
371     /**
372      * Method that determines what happens when pacman collides with a collidable (add
    score,
373      * take out lives, remove from screen, set element in a given location, etc.).
374      */
375     public void removeFromBoard() {
376         double currLocX = pacman.getX() / Constants.SQUARE;
377         double currLocY = pacman.getY() / Constants.SQUARE;
378         if (currLocY < 23 && currLocX > 0) {
379             ArrayList < Collidable > arrayList = boardArray[(int) currLocY][(int)
    currLocX].getArrayList();
380             for (int i = 0; i < arrayList.size(); i++) {
381                 if (arrayList.get(i).isDot() == true) {
382                     this.hearts--;
383                     System.out.println(this.hearts);
```

```
384                     this.controller.addToScore(10);
385                     arrayList.get(i).removeDotFromPane();
386                     arrayList.remove(i);
387                 } else if (arrayList.get(i).isEnergizer() == true) {
388                     this.hearts--;
389                     this.mode = Modes.FRIGHTENED;
390                     this.controller.addToScore(100);
391                     arrayList.get(i).removeEnergizerFromPane();
392                     arrayList.remove(i);
393                 } else if (arrayList.get(i).isGhost() == true) {
394                     gt = 10;
395                     if (this.mode == Modes.FRIGHTENED) {
396                         System.out.println("fkushiewnkf");
397                         int rowP = (int) pacman.getY() / Constants.SQUARE;
398                         int colP = (int) pacman.getX() / Constants.SQUARE;
399                         this.controller.addToScore(200);
400                         this.ghosts.add((Ghost) arrayList.get(i));
401                         ((Ghost) arrayList.get(i)).setPen(300, 275);
402                         this.ghosts.add((Ghost) arrayList.get(i));
403                         boardArray[rowP][colP].getArrayList().remove((Ghost)
    arrayList.get(i));
404                     } else if (this.mode == Modes.CHASE && !(this.mode ==
    Modes.FRIGHTENED) || this.mode == Modes.SCATTER && !(this.mode == Modes.FRIGHTENED)) {
405                         this.pacman.setY(437.5);
406                         this.pacman.setX(287.5);
407                         this.pacman.toFront();
408                         this.controller.addToLives(-1);
409                         boardArray[(int) ghost.getY() / Constants.SQUARE][(int)
    ghost.getX() / Constants.SQUARE].getArrayList().remove(ghost);
410                         boardArray[(int) ghostOne.getY() / Constants.SQUARE][(int)
    ghostOne.getX() / Constants.SQUARE].getArrayList().remove(ghostOne);
411                         boardArray[(int) ghostTwo.getY() / Constants.SQUARE][(int)
    ghostTwo.getX() / Constants.SQUARE].getArrayList().remove(ghostTwo);
412                         boardArray[(int) ghostThree.getY() / Constants.SQUARE][(int)
    ghostThree.getX() / Constants.SQUARE].getArrayList().remove(ghostThree);
413                         this.ghostOne.setPen(300, 200);
414                         this.ghost.setPen(300, 275);
415                         this.ghostTwo.setPen(275, 275);
416                         this.ghostThree.setPen(325, 275);
417                         this.ghosts.add(this.ghost);
418                         this.ghosts.add(this.ghostTwo);
419                         this.ghosts.add(this.ghostThree);
420                     }
421                 }
422             }
423         }
424     }
425     /**
426      * Method that checks if the lives counter is equal to zero (if the game is over).
427      */
428     public void checkGameOver() {
429         if (this.controller.getLives() <= 0) {
430             Label label = new Label();
431             label.setText("GAME OVER");
432             label.setTranslateX(140);
433             label.setTranslateY(250);
434             label.setFont(new Font("Arial", 50));
```

```
435             label.setStyle("-fx-text-fill: white");
436             this.gamePane.getChildren().add(label);
437             this.generalTimeline.stop();
438         }
439     }
440     /**
441      * Method that checks if the lives counter is equal to the maximum number of
    points
442      * (if player won).
443      */
444     public void checkWin() {
445         if (this.hearts == -186) {
446             Label labelWin = new Label();
447             labelWin.setText("YOU WIN!");
448             labelWin.setTranslateX(150);
449             labelWin.setTranslateY(250);
450             labelWin.setFont(new Font("Arial", 50));
451             labelWin.setStyle("-fx-text-fill: white");
452             this.gamePane.getChildren().add(labelWin);
453             this.generalTimeline.stop();
454         }
455     }
456     /**
457      * Method that creates the timeline that holds the continuous pacman movement, the
    collision with
458      * board elements, the wrapping, and checking game over.
459      */
460     private void setUpGeneralTimeline() {
461         KeyFrame keyFrame = new KeyFrame(Duration.seconds(0.5), (ActionEvent) - > {
462             System.out.println(this.mode);
463             this.removeFromBoard();
464             this.pacman.toFront();
465             this.pacman.move();
466             this.removeFromBoard();
467             this.switchModes();
468             this.removeFromBoard();
469             this.pacman.pacmanWrapper();
470             this.ghost.ghostWrapper();
471             this.ghostOne.ghostWrapper();
472             this.ghostTwo.ghostWrapper();
473             this.ghostThree.ghostWrapper();
474             this.checkGameOver();
475             this.checkWin();
476             if (this.gt > 0) {
477                 this.gt--;
478             }
479             if (this.gt == 0) {
480                 if (!(this.ghosts.isEmpty())) {
481                     this.ghosts.remove().setPen(275, 200);
482                     gt = 10;
483                 }
484             }
485         });
486         this.generalTimeline = new Timeline(keyFrame);
487         generalTimeline.setCycleCount(Animation.INDEFINITE);
488         this.generalTimeline.play();
489     }
```

```
    /**
     * Method that creates the timeline that manages and switches the modes of the
ghosts.
     */
    public void switchModes() {
        switch (this.mode) {
            case CHASE:
                this.ghost.setColor(Color.GREEN);
                this.ghostOne.setColor(Color.RED);
                this.ghostTwo.setColor(Color.SKYBLUE);
                this.ghostThree.setColor(Color.YELLOW);
                this.pacman.newColor(Color.YELLOW);
                this.ghostOne.chaseMode(0, 0);
                this.ghostTwo.chaseMode(2, 0);
                this.ghostThree.chaseMode(0, -4);
                this.ghost.chaseMode(-3, 1);
                if (this.chase == 0) {
                    this.mode = Modes.SCATTER;
                    this.scatter = 14;
                } else {
                    this.chase--;
                }
                break;
            case FRIGHTENED:
                this.ghost.randomMovement();
                this.ghostOne.randomMovement();
                this.ghostTwo.randomMovement();
                this.ghostThree.randomMovement();
                this.pacman.newColor(Color.RED);
                this.ghost.setColor(Color.VIOLET);
                this.ghostOne.setColor(Color.VIOLET);
                this.ghostTwo.setColor(Color.VIOLET);
                this.ghostThree.setColor(Color.VIOLET);
                if (this.comeFantasmas == 0) {
                    this.mode = Modes.CHASE;
                    this.chase = 20;
                    this.comeFantasmas = 16;
                } else {
                    this.comeFantasmas--;
                }
                break;
            case SCATTER:
                this.ghostOne.scatterMode(0, 0);
                this.ghostTwo.scatterMode(23, 23);
                this.ghost.scatterMode(0, 23);
                this.ghostThree.scatterMode(23, 0);
                if (this.scatter == 0) {
                    this.mode = Modes.CHASE;
                    this.chase = 20;
                } else {
                    this.scatter--;
                }
                break;
        }
    }
}
package pacman;
```

```
546 import javafx.scene.layout.Pane;
547 import javafx.scene.paint.Color;
548 import java.util.ArrayList;
549 import java.util.LinkedList;
550 import java.util.Queue;
551 /**
552  * Ghost class.
553  */
554 public class Ghost implements Collidable {
555     private Pane gamePane;
556     private int row;
557     private int col;
558     private MazeSquares ghost;
559     private MazeSquares[][] boardArray;
560     private Queue < BoardCoordinate > visited;
561     private Pacman pacman;
562     private Direction directionMoving;
563     private ScoreController controller;
564     /**
565      * Ghost class constructor.
566      */
567     public Ghost(int row, int col, Pane gamePane, MazeSquares[][] boardArray, Pacman
    pacman, ScoreController controller) {
568         this.controller = controller;
569         this.directionMoving = Direction.LEFT;
570         this.pacman = pacman;
571         this.gamePane = gamePane;
572         this.row = row;
573         this.col = col;
574         this.boardArray = boardArray;
575         this.ghost = new MazeSquares(row, col, gamePane);
576         this.ghost.newColor(Color.BLACK);
577     }
578     public void sendToFront() {
579         this.ghost.toFront();
580     }
581     /**
582      * Method allows ghost to change direction if the direction requested is not
    opposite
583      * to the one it had just before and the square to which it will be moving is not
    a wall.
584      */
585     public void changeDirection(Direction dir, MazeSquares[][] squares) {
586         int row = (int) ghost.getYLoc() / Constants.SQUARE;
587         int col = (int) ghost.getXLoc() / Constants.SQUARE;
588         squares[row][col].remove(this);
589         if (!(dir == directionMoving.opposite())) {
590             this.directionMoving = dir;
591             if (!(this.isWall(((int) ghost.getYLoc() / Constants.SQUARE) - 1, (int)
    ghost.getXLoc() / Constants.SQUARE)) && directionMoving == Direction.UP) {
592                 this.ghost.setXLoc(this.ghost.getXLoc());
593                 this.ghost.setYLoc(this.ghost.getYLoc() - Constants.SQUARE);
594             }
595             if (!(this.isWall(((int) ghost.getYLoc() / Constants.SQUARE) + 1, (int)
    ghost.getXLoc() / Constants.SQUARE)) && directionMoving == Direction.DOWN) {
596                 this.ghost.setXLoc(this.ghost.getXLoc());
597                 this.ghost.setYLoc(this.ghost.getYLoc() + Constants.SQUARE);
```

```
598             }
599             if (!(this.isWall((int) ghost.getYLoc() / Constants.SQUARE, ((int)
    ghost.getXLoc() / Constants.SQUARE) + 1)) && directionMoving == Direction.RIGHT) {
600                 this.ghost.setXLoc(this.ghost.getXLoc() + Constants.SQUARE);
601                 this.ghost.setYLoc(this.ghost.getYLoc());
602             }
603             if (!(this.isWall((int) ghost.getYLoc() / Constants.SQUARE, ((int)
    ghost.getXLoc() / Constants.SQUARE) - 1)) && directionMoving == Direction.LEFT) {
604                 this.ghost.setXLoc(this.ghost.getXLoc() - Constants.SQUARE);
605                 this.ghost.setYLoc(this.ghost.getYLoc());
606             }
607         }
608         int nRow = ((int) ghost.getYLoc() / Constants.SQUARE);
609         int nCol = (int) ghost.getXLoc() / Constants.SQUARE;
610         squares[nRow][nCol].add(this);
611     }
612     /**
613      * Method that allows the ghost to move randomly on the board.
614      */
615     public void randomMovement() {
616         if (this.row < 23 && this.row > 0 && this.col > 0 && this.col < 23) {
617             if (!(this.isWall((int) this.ghost.getYLoc() / 25, (int)
    this.ghost.getXLoc() / 25))) {
618                 ArrayList < Direction > directions = validMove();
619                 int number = (int)(Math.random() * directions.size());
620                 this.changeDirection(directions.get(number), boardArray);
621             }
622         }
623     }
624     /**
625      * Arraylist that holds the information about the valid directions the ghost can
    take:
626      * in other words if the squares neighboring the position of the ghost
    corresponding to
627      * a certain direction are not a wall.
628      */
629     public ArrayList validMove() {
630         ArrayList < Direction > dirArray = new ArrayList < > ();
631         if (this.row > 0 && this.row < 24 && this.col > 0 && this.col < 24) {
632             if (!(this.isWall((int) ghost.getYLoc() / Constants.SQUARE, ((int)
    ghost.getXLoc() / Constants.SQUARE) - 1))) {
633                 dirArray.add(Direction.LEFT);
634             }
635             if (!(this.isWall((int) ghost.getYLoc() / Constants.SQUARE, ((int)
    ghost.getXLoc() / Constants.SQUARE) + 1))) {
636                 dirArray.add(Direction.RIGHT);
637             }
638             if (!(this.isWall(((int) ghost.getYLoc() / Constants.SQUARE) - 1, (int)
    ghost.getXLoc() / Constants.SQUARE))) {
639                 dirArray.add(Direction.UP);
640             }
641             if (!(this.isWall(((int) ghost.getYLoc() / Constants.SQUARE) + 1, (int)
    ghost.getXLoc() / Constants.SQUARE))) {
642                 dirArray.add(Direction.DOWN);
643             }
644         }
645         return dirArray;
```

```
646     }
647     /**
648      * Method allows the ghosts to "wrap" from one side of the screen to the other.
649      */
650     public void ghostWrapper() {
651         if (ghost.getXLoc() <= 0) {
652             ghost.setXLoc(550);
653             directionMoving = Direction.LEFT;
654             this.ghost.remove(this);
655         } else if (ghost.getXLoc() >= 550) {
656             ghost.setXLoc(Constants.SQUARE);
657             directionMoving = Direction.RIGHT;
658             this.ghost.remove(this);
659         }
660     }
661     /**
662      * Boolean that detects if a given square on the board is a wall using color
    checking.
663      */
664     public boolean isWall(int row, int col) {
665         if (col >= 0 && col <= 22) {
666             return boardArray[row][col].getColor() != Color.BLACK;
667         }
668         return false;
669     }
670     public boolean getWall(int row, int col) {
671         return isWall(row, col);
672     }
673     /**
674      * Method that checks if the squares neighboring the current location of the
    ghosts are available
675      * (if it's a wall or not), then giving each of the available squares a Direction
    on an array of
676      * Directions and repeating this for the neighbors of the neighbors but this time
    assigning the
677      * neighbors' neighbors the direction of the first neighbor until every spot in
    the array holds a
678      * direction.
679      */
680     public void checkValidNeighbours(Direction dir, Direction[][] dir2DArray) {
681         double ghostCurrLocX = this.ghost.getXLoc() / Constants.SQUARE;
682         double ghostCurrLocY = this.ghost.getYLoc() / Constants.SQUARE;
683         Direction opp = this.directionMoving.opposite();
684         if (col >= 0 && col <= 23) {
685             if (!this.getWall((int) ghostCurrLocY, (int) ghostCurrLocX + 1) && dir ==
    null && opp != Direction.RIGHT && this.row < 23 && this.row > 0 && this.col > 0 &&
    this.col < 23) {
686                 dir2DArray[(int) ghostCurrLocY][(int) ghostCurrLocX + 1] =
    Direction.RIGHT;
687                 visited.add(new BoardCoordinate((int) ghostCurrLocY, (int)
    ghostCurrLocX + 1, false));
688             }
689             if (!this.getWall((int) ghostCurrLocY, (int) ghostCurrLocX - 1) && dir ==
    null && this.directionMoving.opposite() != Direction.LEFT && this.row < 23 && this.row
    > 0 && this.col > 0 && this.col < 23) {
690                 dir2DArray[(int) ghostCurrLocY][(int) ghostCurrLocX - 1] =
    Direction.LEFT;
```

```
691                 visited.add(new BoardCoordinate((int) ghostCurrLocY, (int)
    ghostCurrLocX - 1, false));
692             }
693             if (!this.getWall((int) ghostCurrLocY + 1, (int) ghostCurrLocX) && dir ==
    null && this.directionMoving.opposite() != Direction.DOWN && this.row < 23 && this.row
    > 0 && this.col > 0 && this.col < 23) {
694                 dir2DArray[(int) ghostCurrLocY + 1][(int) ghostCurrLocX] =
    Direction.DOWN;
695                 visited.add(new BoardCoordinate((int) ghostCurrLocY + 1, (int)
    ghostCurrLocX, false));
696             }
697             if (!this.getWall((int) ghostCurrLocY - 1, (int) ghostCurrLocX) && dir ==
    null && this.directionMoving.opposite() != Direction.UP && this.row < 23 && this.row >
    0 && this.col > 0 && this.col < 23) {
698                 dir2DArray[(int) ghostCurrLocY - 1][(int) ghostCurrLocX] =
    Direction.UP;
699                 visited.add(new BoardCoordinate((int) ghostCurrLocY - 1, (int)
    ghostCurrLocX, false));
700             }
701         }
702     }
703     /**
704      * This is the BFS method, that compares the distance of the current square and
    the target using
705      * different directions by analysing the 2DArray of Directions of the
    checkValidNeighbors method.
706      * Once it finds the fastest Direction to get to pacman, it returns that
    Direction.
707      */
708     public Direction BFS(double x, double y) {
709         //make a queue of boardCoordinates
710         int distanceToClosest = 837984;
711         Direction direction = null;
712         this.visited = new LinkedList < > ();
713         Direction[][] dirArray = new Direction[23][23];
714         this.checkValidNeighbours(direction, dirArray);
715         while (!visited.isEmpty()) {
716             BoardCoordinate current = visited.remove();
717             Direction directionCurrent = dirArray[current.getRow()]
    [current.getColumn()];
718             //calculate distance from current to target
719             double distance = Math.hypot(x - current.getColumn(), (y -
    current.getRow()));
720             //add the neighbors of the current square to the queue
721             if (distance < distanceToClosest) {
722                 distanceToClosest = (int) distance;
723                 direction = directionCurrent;
724             }
725         }
726         this.checkValidNeighbours(direction, dirArray);
727         return direction;
728     }
729     /**
730      * Method that allows to change the position of the ghost. Used in the game class
    to take the
731      * ghosts in and out of the pen in appropriate situations.
732      */
```

```
733     public void setPen(double x, double y) {
734         this.ghost.setXLoc(x);
735         this.ghost.setYLoc(y);
736         this.ghost.toFront();
737     }
738     /**
739      * This method uses BFS to make the ghosts move in chaseMove (by setting the
    target to be pacman
740      * or some square close to pacman)
741      */
742     public void chaseMode(int dx, int dy) {
743         this.changeDirection(this.BFS((int) this.pacman.getX() / Constants.SQUARE +
    dx, (int) this.pacman.getY() / Constants.SQUARE + dy), boardArray);
744     }
745     /**
746      * This method uses BFS to make the ghosts move in scatter (by setting the target
    to be one of
747      * the 4 corners of the board)
748      */
749     public void scatterMode(int dx, int dy) {
750         this.changeDirection(this.BFS(dx, dy), boardArray);
751     }
752     /**
753      * Setters and getters.
754      */
755     public double getY() {
756         return this.ghost.getYLoc();
757     }
758     public double getX() {
759         return this.ghost.getXLoc();
760     }
761     public void setX(double x) {
762         this.ghost.setXLoc(x);
763     }
764     public void setY(double y) {
765         this.ghost.setYLoc(y);
766     }
767     public void setColor(Color color) {
768         ghost.newColor(color);
769     }
770     /**
771      * Interface methods.
772      */
773     @Override
774     public void removeDotFromPane() {}
775     @Override
776     public void removeEnergizerFromPane() {}
777     @Override
778     public boolean isEnergizer() {
779         return false;
780     }
781     @Override
782     public boolean isDot() {
783         return false;
784     }
785     /**
786      * Returns true to boolean isGhost.
```

```
787          */
788         @Override
789         public boolean isGhost() {
790             return true;
791         }
792     }
793     package pacman;
794     import javafx.scene.layout.Pane;
795     import javafx.scene.paint.Color;
796     import javafx.scene.paint.Paint;
797     import javafx.scene.shape.Rectangle;
798     import java.util.ArrayList;
799     /**
800      * MazeSquares wrapper class of a Rectangle.
801      */
802     public class MazeSquares {
803         private Rectangle rect;
804         private Pane gamePane;
805         private ArrayList < Collidable > rectArray;
806         private int row;
807         private int col;
808         /**
809          * Constructor of MazeSquares class.
810          */
811         public MazeSquares(int row, int col, Pane gamePane) {
812             this.rectArray = new ArrayList < Collidable > ();
813             this.gamePane = gamePane;
814             this.row = row;
815             this.col = col;
816             this.rect = new Rectangle(col * Constants.SQUARE, row * Constants.SQUARE,
    Constants.SQUARE, Constants.SQUARE);
817             this.gamePane.getChildren().add(this.rect);
818         }
819         /**
820          * Setters and getters.
821          */
822         public void toFront() {
823             this.rect.toFront();
824         }
825         public ArrayList getArrayList() {
826             return rectArray;
827         }
828         public void setYLoc(double y) {
829             this.rect.setY(y);
830         }
831         public void setXLoc(double x) {
832             this.rect.setX(x);
833         }
834         public double getYLoc() {
835             return this.rect.getY();
836         }
837         public double getXLoc() {
838             return this.rect.getX();
839         }
840         public void newColor(Color color) {
841             this.rect.setFill(color);
842         }
```

```
    public Paint getColor() {
        return this.rect.getFill();
    }
    public void remove(Collidable collidable) {
        this.rectArray.remove(collidable);
    }
    public void add(Collidable collidable) {
        this.rectArray.add(collidable);
    }
}
package pacman;
/**
 * Modes enum class.
 */
public enum Modes {
    CHASE,
    SCATTER,
    FRIGHTENED;
}
package pacman;
import javafx.scene.input.KeyCode;
import javafx.scene.input.KeyEvent;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
/**
 * Pacman class.
 */
public class Pacman {
    private Circle pacman;
    private Pane gamePane;
    private boolean isLeft;
    private boolean isUp;
    private boolean isDown;
    private boolean isMovingLeft;
    private boolean isMovingUp;
    private boolean isMovingDown;
    private MazeSquares[][] boardArray;
    /**
     * Constructor of the Pacman class.
     */
    public Pacman(Pane gamePane, MazeSquares[][] boardArray) {
        isLeft = true;
        isUp = true;
        isDown = true;
        isMovingLeft = true;
        isMovingUp = true;
        isMovingDown = true;
        this.boardArray = boardArray;
        this.gamePane = gamePane;
        this.pacman = new Circle(287.5, 437.5, 12.5);
        this.pacman.setFill(Color.YELLOW);
        pacman.setOnKeyPressed((KeyEvent e) - > this.handleKeyPress(e));
        pacman.setFocusTraversable(true);
        this.gamePane.getChildren().add(pacman);
    }
    /**
```

```
900         * Method that sends pacman to the front of the pane.
901         */
902        public void toFront() {
903            this.pacman.toFront();
904        }
905        public void setBoardArray(MazeSquares[][] maze) {
906            this.boardArray = maze;
907        }
908        /**
909         * Method allows pacman to "wrap" from one side of the screen to the other.
910         */
911        public void pacmanWrapper() {
912            if (pacman.getCenterX() <= 12.5) {
913                pacman.setCenterX(537.5);
914            } else if (pacman.getCenterX() >= 562.5) {
915                pacman.setCenterX(37.5);
916            }
917        }
918        /**
919         * Method that manages what should happen when an arrow is pressed on the
    keyboard.
920         */
921        private void handleKeyPress(KeyEvent e) {
922            KeyCode keyPressed = e.getCode();
923            switch (keyPressed) {
924                case LEFT:
925                    if (canMove(-1, 0)) {
926                        isLeft = true;
927                        isUp = false;
928                        isDown = false;
929                    }
930                    break;
931                case RIGHT:
932                    if (canMove(1, 0)) {
933                        isLeft = false;
934                        isUp = false;
935                        isDown = false;
936                    }
937                    break;
938                case UP:
939                    if (canMove(0, -1)) {
940                        isLeft = false;
941                        isUp = true;
942                        isDown = false;
943                    }
944                    break;
945                case DOWN:
946                    if (canMove(0, 1)) {
947                        isUp = false;
948                        isLeft = false;
949                        isDown = true;
950                    }
951                    break;
952                default:
953                    break;
954            }
955            e.consume();
```

```
956         }
957         /**
958          * Boolean that detects if a given square on the board is a wall using color
    checking.
959          */
960         public boolean isWall(int row, int col) {
961             if (col <= 22 && col >= 0) {
962                 return boardArray[row][col].getColor() != Color.BLACK;
963             }
964             return false;
965         }
966         /**
967          * Boolean that determines if pacman can move by checking if there is a wall in
    the
968          * square it wants to move to.
969          */
970         public boolean canMove(int dx, int dy) {
971             if (this.isWall((int) pacman.getCenterY() / Constants.SQUARE + dy, (int)
    pacman.getCenterX() / Constants.SQUARE + dx)) {
972                 return false;
973             }
974             return true;
975         }
976         /**
977          * Method called in the generalTimeline in the game class that manages how pacman
    should move
978          * using the booleans used inside the handleKeyPress method.
979          */
980         public void move() {
981             //case Left
982             if (isLeft == true && isUp == false && isDown == false) {
983                 if (this.canMove(-1, 0)) {
984                     pacman.setCenterX(pacman.getCenterX() - Constants.SQUARE);
985                     this.isMovingLeft = true;
986                     this.isDown = false;
987                     this.isMovingUp = false;
988                 } else {
989                     if (isMovingUp && !isMovingDown && !isMovingLeft && this.canMove(0,
    -1)) {
990                         pacman.setCenterY(pacman.getCenterY() - Constants.SQUARE);
991                     } else if (!isMovingUp && isMovingDown && !isMovingLeft &&
    this.canMove(0, 1)) {
992                         pacman.setCenterY(pacman.getCenterY() + Constants.SQUARE);
993                     }
994                 }
995                 // case Right
996             } else if (isLeft == false && isUp == false && isDown == false) {
997                 if (this.canMove(1, 0)) {
998                     pacman.setCenterX(pacman.getCenterX() + Constants.SQUARE);
999                     this.isMovingLeft = false;
1000                    this.isMovingUp = false;
1001                    this.isMovingDown = false;
1002                } else {
1003                    if (isMovingUp && !isMovingDown && !isMovingLeft && this.canMove(0,
    -1)) {
1004                        pacman.setCenterY(pacman.getCenterY() - Constants.SQUARE);
```

```
                } else if (!isMovingUp && isMovingDown && !isMovingLeft &&
this.canMove(0, 1)) {
                    pacman.setCenterY(pacman.getCenterY() + Constants.SQUARE);
                }
            }
            //case Up
        } else if (isLeft == false && isUp == true && isDown == false) {
            if (this.canMove(0, -1)) {
                pacman.setCenterY(pacman.getCenterY() - Constants.SQUARE);
                this.isMovingLeft = false;
                this.isMovingUp = true;
                this.isMovingDown = false;
            } else {
                if (!isMovingUp && !isMovingDown && isMovingLeft && this.canMove(-1,
0)) {
                    pacman.setCenterX(pacman.getCenterX() - Constants.SQUARE);
                } else if (!isMovingUp && !isMovingDown && !isMovingLeft &&
this.canMove(1, 0)) {
                    pacman.setCenterX(pacman.getCenterX() + Constants.SQUARE);
                }
            }
            //case Down
        } else if (!isLeft && !isUp && isDown) {
            if (this.canMove(0, 1)) {
                pacman.setCenterY(pacman.getCenterY() + Constants.SQUARE);
                this.isMovingLeft = false;
                this.isMovingUp = false;
                this.isMovingDown = true;
            } else {
                if (!isMovingUp && !isMovingDown && isMovingLeft && this.canMove(-1,
0)) {
                    pacman.setCenterX(pacman.getCenterX() - Constants.SQUARE);
                } else if (!isMovingUp && !isMovingDown && !isMovingLeft &&
this.canMove(1, 0)) {
                    pacman.setCenterX(pacman.getCenterX() + Constants.SQUARE);
                }
            }
        }
    }
    /**
     * Setters and getters.
     */
    public double getX() {
        return this.pacman.getCenterX();
    }
    public double getY() {
        return this.pacman.getCenterY();
    }
    public void setX(double x) {
        this.pacman.setCenterX(x);
    }
    public void setY(double y) {
        this.pacman.setCenterY(y);
    }
    public void newColor(Color color) {
        this.pacman.setFill(color);
    }
```

```
}
package pacman;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Pane;
/**
 * PaneOrganizer class.
 */
public class PaneOrganizer {
    private BorderPane root;
    private Pane gamePane;
    public MazeSquares[][] boardArray;
    private MazeSquares square;
    /**
     * PaneOrganizer constructor.
     */
    public PaneOrganizer() {
        this.root = new BorderPane();
        this.gamePane = new Pane();
        this.boardArray = new MazeSquares[23][23];
        HBox quitBox = new HBox();
        quitBox.setStyle("-fx-background-color: WHITE");
        quitBox.setPrefSize(400, 60);
        new Game(gamePane, boardArray, new ScoreController(quitBox), square);
        this.root.setCenter(gamePane);
        this.root.setBottom(quitBox);
    }
    public BorderPane getRoot() {
        return root;
    }
}
package pacman;
import javafx.event.ActionEvent;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.HBox;
/**
 * ScoreController class.
 */
public class ScoreController {
    private int score;
    private int lives;
    private Label scoreLabel;
    private Label livesLabel;
    /**
     * ScoreController constructor.
     */
    public ScoreController(HBox pane) {
        Button quit = new Button("Quit");
        quit.setOnAction((ActionEvent e) - > System.exit(0));
        this.score = 0;
        this.scoreLabel = new Label("                                    Score: " +
0);
        this.lives = 3;
        this.livesLabel = new Label("                                    Lives: " +
3);
        pane.getChildren().addAll(quit, this.scoreLabel, this.livesLabel);
```

```
1112        }
1113        /**
1114         * Method that allows to add points to the score on the screen.
1115         */
1116        public void addToScore(int points) {
1117            this.score += points;
1118            this.scoreLabel.setText("                                    Score: " +
    this.score);
1119        }
1120        /**
1121         * Method that allows to add or remove lives on the screen.
1122         */
1123        public void addToLives(int hearts) {
1124            this.lives += hearts;
1125            this.livesLabel.setText("                                    Lives: " +
    this.lives);
1126        }
1127        /**
1128         * Getter for the number of lives.
1129         */
1130        public int getLives() {
1131            return this.lives;
1132        }
1133        /**
1134         * Getter for the number of points.
1135         */
1136        public int getScore() {
1137            return this.score;
1138        }
1139    }
```