Matthew Mendoza

Homework 03 – PongAPP & OPongAPP

0319

APPENDIX

PongApp.java – PONG 1.0

```
import javafx.animation.AnimationTimer;
import javafx.application.Application;
import javafx.event.EventHandler;
import javafx.scene.control.Label;
import javafx.scene.Cursor;
import javafx.scene.Group;
import javafx.scene.input.KeyCode;
import javafx.scene.input.KeyEvent;
import javafx.scene.input.MouseEvent;
import javafx.scene.paint.Color;
import javafx.scene.Scene;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Font;
import javafx.stage.Stage;
import javax.sound.midi.Instrument;
import javax.sound.midi.MidiChannel;
import javax.sound.midi.MidiSystem;
import javax.sound.midi.MidiUnavailableException;
import javax.sound.midi.Synthesizer;


/**
 * Now that you're up to speed on JavaFX and the basic linear style, you
will
 * want to code your first version of the video game. In the Asteroids game
 * the author uses a simple class hierarchy for his game objects where all
 * game objects extend the abstract class PhysicsObject. For this first
 * version of Pong, it's not necessary to create a class
 * hierarchy as the objects are so simple.
```

```
 *
 * You may simply encode them as fields in the main class, e.g.:
 * Rectangle paddle = new Rectangle(...);
 * Rectangle ball = new Rectangle(...);
 *
 * Of course you can separate declaration from initialization if you wish.
 *
 * The purpose of this first project is for you to code freely and solve all
 * of the simple algorithmic problems such as paddle/ball/wall intersection.
 * You do not have to write the sound class at all, it is provided in the
 * appendix. Your first version should be as close as possible to the given
 * demo.
 *
 * Your application should respond to two keystroke events:
 * - The “i” key will display or hid the fps information and
 * - the “s” key will enable and disable sound.
 */
public class PongApp extends Application {

    /**
     * Shrinks the paddle by this amount
     */
    private static final int PADDLE_SHRINK_FACTOR = getPADDLE_W() - 10;

    /**
     *
     */
    private static final Color SCORE_INFO_COLOR = Color.BLUE;

    /**
     * The position of the FPS information label in the scene.
```

```
     * This value is used for both the x and y coordinates.
     */
    private static final int FPS_DISPLAY_INFO_LABEL_X_Y_POSITION = 10;

    /**
     * In-game timer string format to display the number of frames per
second.
     */
    private static final String GAME_TIME_INFO_STRING = "GT: %.2f (s)";

    /**
     * Frame time string format to display the number of frames per
     * millisecond.
     */
    private static final String FRAME_TIME_INFO_STRING = "FT: %.2f (ms)";

    /**
     * The Frames Per Second (FPS) to be displayed.
     */
    private static final String FPS_INFO_STRING = "FPS: %.2f (avg.)";

    /**
     * Game information label is the concatenation of the FPS, frame time,
and
     * in-game timer.
     */
    private static final String FPS_DISPLAY_INFO_STRING = FPS_INFO_STRING +
            " , " +
            FRAME_TIME_INFO_STRING +
            " , " +
            GAME_TIME_INFO_STRING;
```

```
/**
 * 1_000.0 is the number of nanoseconds in a millisecond
 */
private static final double TIME_01_MILLISECOND = 1_000.0;

/**
 * 1_000_000_000.0 nanoseconds = 1 second
 */
private static final double TIME_01_SECOND = 1_000_000_000.0;
private static final String APP_TITLE = "Pong 1.0";
private static final int APP_H = 600;
private static final int APP_W = 800;
private static final String APP_FONT = "Arial";
private static final Color FPS_INFO_COLOR = Color.BLACK;
private static final int APP_FONT_SIZE_24 = 24;
private static boolean newGame = true;
private static double APP_H_Y_CORD_1_3RD_FROM_TOP = APP_H / 3;

private static double avgFpMiliSecond = 0;
private static double avgFPS = 0;
private static double secondsElapsedInGame = 0;
private static final int NUMBER_OF_FRAMES_25 = 25;
private static double[] frameRateArr = new double[NUMBER_OF_FRAMES_25];
private static int animationTimerFrameCounter = 0;
private static long lastTime = 0;
private static long startTime = System.nanoTime();
private static int gameScoreCounter = 0;

private static boolean isGameScore50 = false;

private static double crntCrsrX = 0;
```

```
private static double prevCrsrX = 0;

private static final int BALL_H = 20;
private static final int BALL_W = BALL_H;
private static boolean isBallUp = false;
private static boolean isBallLeft = false;
private static boolean isBallRotLeft = false;
private static final int BALL_SPEED_Y_MIN = 5;
private static final int BALL_X_CORD_MAX_RESPAWN_LIMIT = APP_W - BALL_W;
private static final int BALL_X_CORD_MIN_RESPAWN_LIMIT = 0;
private static int ballRotSpd = 0;
private static int ballRotVel = 0;
private static int ballSpdX = 0;
private static int ballSpdY = 0;
private static int ballVelX = 0;
private static int ballVelY = 0;

private static final int PADDLE_H = 20;
private static int PADDLE_W = 150;

private static boolean isPaddleLeft = false;
private static boolean isPaddleStationary = true;
private static boolean isPaddleAboveMinimalSizeCap = false;
private static int paddleMinimalSizeCap = 50;
private static int paddleSpdX = 0;
private static int paddleVelX = 0;

public static void setGameScoreCounter(int gameScoreCounter) {
    PongApp.gameScoreCounter = gameScoreCounter;
}
```

```
public static int getPaddleMinimalSizeCap() {
    return paddleMinimalSizeCap;
}

public static void isGameScore50() {
    if ((getGameScoreCounter() >= 50)) {
        setGameScore50(true);
    } else {
        setGameScore50(false);
    }
}

public static void setGameScore50(boolean isGameScore50) {
    PongApp.isGameScore50 = isGameScore50;
}

public static boolean isPaddleAboveMinimalSizeCap() {;
    return (getPADDLE_W() < getPaddleMinimalSizeCap()) ? true : false;
}

public static int getPADDLE_W() {
    return PADDLE_W;
}

public static void setPADDLE_W(int pADDLE_W) {
    PADDLE_W = pADDLE_W;
}

public static void shrinkPaddle() {
    if (isPaddleAboveMinimalSizeCap()) {
        setPADDLE_W(PADDLE_SHRINK_FACTOR);
```

```
    }
}

public static int getGameScoreCounter() {
    return gameScoreCounter;
}

public static void incrementGameScoreCounter() {
    PongApp.gameScoreCounter += 1;
}

public static boolean isNewGame() {
    return newGame;
}

public static void setNewGame(boolean newGame) {
    PongApp.newGame = newGame;
}

/**
 * Direction of Ball object to be factored in Ball's velocity
 *
 * @return true if ball is moving left, false if ball is moving right
 */
public static boolean isBallUp() {
    return isBallUp;
}

/**
 * Direction of Ball object to be factored in Ball's velocity
 *
```

```
 * @param isBallUp true if ball is moving up, false otherwise
 */
public static void setBallUp(boolean isBallUp) {
    PongApp.isBallUp = isBallUp;
}

/**
 * Direction of Ball object to be factored in Ball's velocity
 *
 * @return true if ball is moving left, false if ball is moving right
 */
public static boolean isBallLeft() {
    return isBallLeft;
}

/**
 * Direction of Ball object to be factored in Ball's velocity
 *
 * @param isBallUp true if ball is moving left, false otherwise
 */
public static void setBallLeft(boolean isBallLeft) {
    PongApp.isBallLeft = isBallLeft;
}

public static boolean isBallRotLeft() {
    return isBallRotLeft;
}

public static void setBallRotLeft(boolean isBallRotLeft) {
    PongApp.isBallRotLeft = isBallRotLeft;
}
```

```
public static int getBallRotSpd() {
    return ballRotSpd;
}

public static void setBallRotSpd(int ballRotSpd) {
    PongApp.ballRotSpd = ballRotSpd;
}

public static int getBallRotVel() {
    return ballRotVel;
}

/**
 * Calculates the ball's rotational velocity based on the ball's
 * rotational speed and direction of rotation
 *
 * @param ballRotVel the ball's rotational velocity
 */
public static void calculateBallRotVel() {
    PongApp.ballRotVel = (isBallRotLeft())
            ? -getBallRotSpd() // if ball is rotating left, then
                               // ballRotVel is negative
            : getBallRotSpd(); // if ball is rotating right, then
                               // ballRotVel is positive
}

/**
 * Speed is the time rate at which the ball is moving along a path
 *
 * @return the speed of the ball in the x direction
```

```
 */
public static int getBallSpdX() {
    return ballSpdX;
}

/**
 * Speed is the time rate at which the ball is moving along a path
 *
 * @param ballSpdX the speed of the ball in the x direction
 */
public static void setBallSpdX(int ballSpdX) {
    PongApp.ballSpdX = ballSpdX;
}

public static int getBallSpdY() {
    return ballSpdY;
}

public static void setBallSpdY(int ballSpdY) {
    PongApp.ballSpdY = ballSpdY;
}

/**
 * velocity is the rate (speed) and
 * direction (isBallLeft) of the ball's movement
 *
 * @return the velocity of the ball in the x direction
 */
public static int getBallVelX() {
    return ballVelX;
}
```

```
/**
 * Calculates ball's velocity based on its rate (speed) and
 * direction (isBallLeft) movement
 */
public static void calculateBallVelX() {
    PongApp.ballVelX = (isBallLeft()
            ? -getBallSpdX() // if ball is moving left,
                             // then ballVelX is negative
            : getBallSpdX()); // if ball is moving right,
                              // then ballVelX is positive
}

/**
 * velocity is the rate (speed) and
 * direction (isBallUp) of the ball's movement
 *
 * @return the velocity of the ball in the y direction
 */
public static int getBallVelY() {
    return ballVelY;
}

/**
 * Calculates ball's velocity based on its rate (speed) and
 * direction (isBallUp) movement
 */
public static void calculateBallVelY() {
    PongApp.ballVelY = (isBallUp() ? -getBallSpdY() : getBallSpdY());
}
```

```
public static int getPaddleSpdX() {
    return paddleSpdX;
}

/**
 * Calculates paddle's speed based on the distance between
 * current and previous cursor x coordinates
 */
public static void calculatePaddleSpdX() {
    PongApp.paddleSpdX = Math.abs((int) ((getCrntCrsrX()
            - getPrevCrsrX()) // distance between current and previous
                              // cursor x coordinates
            / NUMBER_OF_FRAMES_25)); // divided by number of frames
                                     // per second
}

public static int getPaddleVelX() {
    return paddleVelX;
}

/**
 * Calculate the paddle's velocity based on the cursor's
 * previous and current position over time
 */
public static void calculatePaddleVelX() {
    // calculate paddle's speed to be used in calculating its velocity
    calculatePaddleSpdX();
    PongApp.paddleVelX = (isPaddleLeft()) // if paddle is moving left
            ? -getPaddleSpdX() // then paddleVelX is negative
            : getPaddleSpdX(); // if paddle is moving right
                               // then paddleVelX is positive
```

```
}

public static double getPrevCrsrX() {
    return prevCrsrX;
}

public static void setPrevCrsrX(double prevCrsrX) {
    PongApp.prevCrsrX = prevCrsrX;
}

public static double getCrntCrsrX() {
    return crntCrsrX;
}

public static void setCrntCrsrX(double crntCrsrX) {
    PongApp.crntCrsrX = crntCrsrX;
}

public static boolean isPaddleLeft() {
    return isPaddleLeft;
}

public static void setPaddleLeft(boolean isPaddleLeft) {
    PongApp.isPaddleLeft = isPaddleLeft;
}

public static boolean isPaddleStationary() {
    return isPaddleStationary;
}

public static void setPaddleStationary(boolean isPaddleStationary) {
```

```
        PongApp.isPaddleStationary = isPaddleStationary;
    }

    public static double getAvgFpMiliSecond() {
        return avgFpMiliSecond;
    }

    public static void setAvgFpMiliSecond(double avgFpMiliSecond) {
        PongApp.avgFpMiliSecond = avgFpMiliSecond;
    }

    public double getAvgFPS() {
        return avgFPS;
    }

    public static void setAvgFPS(double avgFPS) {
        PongApp.avgFPS = avgFPS;
    }

    public static double getSecondsElapsedInGame() {
        return secondsElapsedInGame;
    }

    public static void setSecondsElapsedInGame(double secondsElapsedInGame)
{
        PongApp.secondsElapsedInGame = secondsElapsedInGame;
    }

    public static int getAnimationTimerFrameCounter() {
        return animationTimerFrameCounter;
    }
```

```
    public static void setAnimationTimerFrameCounter(int aniTimerFrmCtr) {
        PongApp.animationTimerFrameCounter = aniTimerFrmCtr;
    }

    public static long getStartTime() {
        return startTime;
    }

    /**
     * Defines the mouse/courser movements to
     * control the paddle movement and color.
     *
     * @param scene  the scene of the game to add the event handler
     * @param paddle the paddle to move and change color
     */
    private void paddleMouseMovementController(Scene scene, Rectangle
paddle) {

        /**
         * the paddle is blue when the mouse enters the scene
         */
        scene.setOnMouseEntered(new EventHandler<MouseEvent>() {
            @Override
            public void handle(MouseEvent event) {
                paddle.setFill(SCORE_INFO_COLOR);
                // Mouse courser disappears when it enters the scene
                scene.setCursor(Cursor.NONE);
            }
        });
```

```
        /**
         * The paddle color is red when the mouse is outside the game
window.
         */
        scene.setOnMouseExited(new EventHandler<MouseEvent>() {
            @Override
            public void handle(MouseEvent event) {
                paddle.setFill(Color.RED);
            }
        });

        /**
         * The paddle moves left and right when the user's mouse/courser
         * enters the game window
         */
        scene.setOnMouseMoved(new EventHandler<MouseEvent>() {
            @Override
            public void handle(MouseEvent event) {
                // So paddle doesn't go off screen to the left
                double PADDLE_X_MIN = 0;
                // So paddle doesn't go off screen to the right
                double PADDLE_X_MAX = APP_W - paddle.getWidth();
                paddle.setFill(SCORE_INFO_COLOR);

                // Move the paddle center to the mouse position
                setCursorToCenterOfPaddle(paddle, event);
                // Make sure the paddle stays inside the game window
                setPaddleInGameBndry(paddle, PADDLE_X_MIN, PADDLE_X_MAX);
                // Grab the mouse position
                setCrntCrsrX(event.getSceneX());
                calculatePaddleVelX();
```

```
        }

        /**
         * Sets the paddle in the game window boundaries.
         * If the paddle is at the left or right edge of the game
window,
         * the paddle will not move.
         *
         * @param paddle       the paddle to set in the game window
         * @param PADDLE_X_MIN the minimum x-coordinate of the paddle
         * @param PADDLE_X_MAX the maximum x-coordinate of the paddle
         */
        private void setPaddleInGameBndry(
                Rectangle paddle,
                double PADDLE_X_MIN,
                double PADDLE_X_MAX) {
            // left edge of game window is
            if (paddle.getTranslateX() < PADDLE_X_MIN) {
                paddle.setTranslateX(PADDLE_X_MIN);
            }
            // right edge of game window
            else if (paddle.getTranslateX() > PADDLE_X_MAX) {
                paddle.setTranslateX(PADDLE_X_MAX);
            }
        }

        /**
         * Sets the mouse cursor to the center of the paddle.
         *
         * @param paddle the paddle to set the mouse cursor
         * @param event  the mouse event to get the mouse cursor
location
```

```
         */
        private void setCursorToCenterOfPaddle(
                Rectangle paddle,
                MouseEvent event) {
            paddle.setTranslateX(event.getX() - paddle.getWidth() / 2);
        }
    });
}

/**
 * The game loop is a loop that runs until game window is closed.
 * In each iteration of the loop:
 *
 * The game state is updated and the game is rendered.
 *
 * The loop runs at a fixed rate, which means that the game runs at
 * the same speed on all computers.
 *
 * The game loop is started when the game window is shown.
 * The game loop is stopped when the game window is closed.
 *
 * Respond to two keystroke events:
 *
 * The “i” key will display or hide the FPS information and
 * the “s” key will enable and disable sound effects.
 *
 * @param primaryStage the stage to be shown when the application is
 *                     started
 */
@Override
public void start(Stage primaryStage) throws Exception {
```

```
Group root = new Group();
Scene scene = new Scene(root, APP_W, APP_H);
scene.setFill(Color.WHITE);
primaryStage.setResizable(false);
primaryStage.setScene(scene);
primaryStage.setTitle(APP_TITLE);
primaryStage.show();

// Labels for Frames Per Second (FPS) information
Label fpsDisplayInfoLabel = new Label("FPS: ");
fpsDisplayInfoLabel.setFont(new Font(APP_FONT, APP_FONT_SIZE_24));
fpsDisplayInfoLabel.setTextFill(FPS_INFO_COLOR);
fpsDisplayInfoLabel.setLayoutX(FPS_DISPLAY_INFO_LABEL_X_Y_POSITION);
fpsDisplayInfoLabel.setLayoutY(FPS_DISPLAY_INFO_LABEL_X_Y_POSITION);
// add the labels to the scene graph
root.getChildren().add(fpsDisplayInfoLabel);

// Labels score information
Label scoreDisplayInfoLabel = new Label("Score: ");
scoreDisplayInfoLabel.setFont(new Font(APP_FONT, APP_FONT_SIZE_24));
scoreDisplayInfoLabel.setTextFill(SCORE_INFO_COLOR);
// set score label position in the middle of the game window
scoreDisplayInfoLabel.setLayoutX(
        (APP_W / 2) - (scoreDisplayInfoLabel.getWidth() / 2));
scoreDisplayInfoLabel.setLayoutY(APP_H_Y_CORD_1_3RD_FROM_TOP);
// add the labels to the scene graph
root.getChildren().add(scoreDisplayInfoLabel);

/**
 * Paddle objects are rectangles that move up and down the screen.
 */
```

```
Rectangle paddle = new Rectangle(PADDLE_W, PADDLE_H);
paddle.setFill(Color.RED);
paddle.setTranslateY(APP_H - paddle.getHeight());
root.getChildren().add(paddle);

/**
 * Ball objects are rectangles that move around the screen.
 */
Rectangle ball = new Rectangle(BALL_W, BALL_H);
ball.setFill(SCORE_INFO_COLOR);
// random x-coordinate for ball object
ball.setTranslateX((Math.random() * (APP_W - ball.getWidth())));
ball.setTranslateY(APP_H_Y_CORD_1_3RD_FROM_TOP);
root.getChildren().add(ball);

/**
 * BinkBonkSound objects are sound effects that play when the ball
 */
BinkBonkSound sound = new BinkBonkSound();

////////////////////////////
// Key events for sound
// effects and fps display
////////////////////////////
/**
 * Respond to two keystroke events:
 * The “i” key will display or hide the FPS information and
 * the “s” key will enable and disable sound effects.
 */
scene.setOnKeyPressed(new EventHandler<KeyEvent>() {
    @Override
```

```
    public void handle(KeyEvent event) {
        if (event.getCode() == KeyCode.I) {
            // Toggle FPS display information
            fpsDisplayInfoLabel.setVisible(
                    !fpsDisplayInfoLabel.isVisible());
        } else if (event.getCode() == KeyCode.S) {
            // Toggle sound
            sound.toggleSound();
        }
    }
});
////////////////////////////
// Mouse events for paddle
////////////////////////////

/**
 * Respond to mouse movement events.
 * The paddle's center is set to the mouse's x coordinate.
 * The paddle will not go off the screen (stays within the window).
 *
 * When the mouse is moved off the screen
 * - the paddle will stop moving
 * - stay in the last position it was in
 * - change color to red.
 *
 * When the mouse is moved back on the screen
 * - the paddle will move to the latest cursor position
 * - the paddle's middle will center on the cursor's position
 */
paddleMouseMovementController(scene, paddle);
```

```
        /**
         * The game loop is a loop that runs until game window is closed.
         */
        AnimationTimer timer = new AnimationTimer() {

            /**
             * ALL THE MAGIC HAPPENS HERE IN THE GAME LOOP METHOD
             *
             * The game loop is a loop that runs until game window is
closed.
             * In each iteration of the loop:
             * - The game state is updated and the game is rendered.
             * - The loop runs at a fixed rate, which means that the game
             * runs at the same speed on all computers.
             * - The game loop is started when the game window is shown.
             * - The game loop is stopped when the game window is closed.
             *
             * @param now the current time in nanoseconds
             *            (1 billionth of a second)
             */
            @Override
            public void handle(long now) {
                checkStartOfNewGame(); // Reset to a new game state
                updatePrevCrsrLocation(); // update previous cursor location
                updatePaddleMovement(); // paddle movement (left, right,
stop)
                updateBallMovement(); // ball X & Y and rotational velocity
                ballHitsPaddle(); // Logic for ball to paddle collision
                ballHitsScrBndry(); // Logic for ball to screen collision
                setInGameTimeAndAvgFrameTimeAndFPS(now); // calculate FPS
                updateFPSDisplayInformation(); // update FPS display info
                updateScoreDisplay(); // update score display info
```

```
    if (getGameScoreCounter() > 2) {
        PADDLE_W -= (int) (getPADDLE_W() / 2);
    }
    animationTimerFrameCounter++; // increment the frame counter
}

private void updateScoreDisplay() {
    scoreDisplayInfoLabel.setText(
            String.format("%d",
                    getGameScoreCounter()));
}

/**
 * Every new game the ball only moves vertically,
 * no velocity in the x-axis, along the y-axis down the
 * screen
 */
private void checkStartOfNewGame() {
    if (newGame) {
        respawnBallToRandomLocation();
        setBallUp(false); // ball only moves vertically
        setBallLeft(false); // only moves vertically
        setBallSpdX(0); // no velocity in the x-axis
        setBallSpdY(BALL_SPEED_Y_MIN); // along the y-axis down
        setNewGame(false); // game has started
        setBallRotSpd(0);
        ball.setRotate(0);
        setGameScoreCounter(0); // reset score
    }
}
```

```
/**
 * every 25 frames, assign
 * current cursor x position as
 * previous cursor x position
 */
private void updatePrevCrsrLocation() {
    if (getAnimationTimerFrameCounter()
            % NUMBER_OF_FRAMES_25 == 0) {
        setPrevCrsrX(getCrntCrsrX());
    }
}

/**
 * compares current cursor position to its previous position
 * to determine if the paddle is moving left, right,
 * or not at all (stationary)
 */
private void updatePaddleMovement() {
    paddleIsMovingRight();
    paddelIsMovingLeft();
    paddleIsStationary();
}

/**
 * moving right when the cursor is to the right of the paddle
 */
private void paddleIsMovingRight() {
    if (getCrntCrsrX() > getPrevCrsrX()) {
        // moving right
        setPaddleLeft(false);
    }
```

```
}

/**
 * moving left when the cursor is to the left of the paddle's
 * center
 */
private void paddelIsMovingLeft() {
    if (getCrntCrsrX() < getPrevCrsrX()) {
        // moving left
        setPaddleLeft(true);
    }
}

/**
 * stationary when the cursor is not moving
 */
private void paddleIsStationary() {
    if (getCrntCrsrX() == getPrevCrsrX()) {
        // stationary
        setPaddleStationary(true);
    }
}

private void ballHitsPaddle() {
    if (ball.getBoundsInParent()
            .intersects(paddle.getBoundsInParent())) {
        // Ball hits the paddle bounce off the paddle
        setBallUp(true);
        // ball's x velocity is equal to its y velocity
        setBallSpdX(getBallSpdY());
        ballHitsLeftMovingPaddle();
```

```
            ballHitsRightMovingPaddle();
            ballHitsStationaryPaddle();
            shrinkPaddle();
            sound.play(true);
            incrementGameScoreCounter();
        }
    }

    /**
     * If the ball hits the paddle when the paddle is moving left
     * the ball will move bounce left and rotate right
     */
    private void ballHitsLeftMovingPaddle() {
        if (getCrntCrsrX() < getPrevCrsrX()) {
            setBallLeft(true);
            setBallRotLeft(true);
            setBallRotSpd(getPaddleSpdX());
        }
    }

    /**
     * If the ball hits the paddle when the paddle is moving right
     * the ball will bounce right and rotate left
     */
    private void ballHitsRightMovingPaddle() {
        if (getCrntCrsrX() > getPrevCrsrX()) {
            setBallLeft(false);
            setBallRotLeft(false);
            setBallRotSpd(getPaddleSpdX());
        }
    }
```

```
/**
 * if the ball hits the paddle when the paddle is stationary
 * the ball will move bounce up and have no rotation
 */
private void ballHitsStationaryPaddle() {
    if (getCrntCrsrX() == getPrevCrsrX()) {
        setBallSpdX(0);
    }
}

/**
 * A wrapper method for the ball's interaction with
 * the screen boundaries (top, bottom, left, right).
 */
private void ballHitsScrBndry() {
    // top of screen
    ballHitsTopOfScreenBounceDown();
    // left side of screen
    BallHitsLeftOfScreenBounceRight();
    // right side of screen
    ballHitsRightOfScreenBounceLeft();
    // game resets when ball hits the bottom of the screen
    ballHitsBottomOfScreenResetGame();
}

private void ballHitsTopOfScreenBounceDown() {
    if (ball.getTranslateY() <= 0) {
        setBallUp(false);
        sound.play(false);
        incrementGameScoreCounter();
```

```
            setBallSpdX(getBallSpdX() + 1);
            setBallSpdY(getBallSpdY() + 1);
        }

    }

    private void BallHitsLeftOfScreenBounceRight() {
        if (ball.getTranslateX() <= BALL_X_CORD_MIN_RESPAWN_LIMIT) {
            setBallLeft(false);
            sound.play(false);
            incrementGameScoreCounter();
            setBallSpdX(getBallSpdX() + 1);
            setBallSpdY(getBallSpdY() + 1);
        }
    }

    private void ballHitsRightOfScreenBounceLeft() {
        if (ball.getTranslateX() >= BALL_X_CORD_MAX_RESPAWN_LIMIT) {
            // bounce off right side of screen
            setBallLeft(true);
            sound.play(false);
            incrementGameScoreCounter();
            setBallSpdX(getBallSpdX() + 1);
            setBallSpdY(getBallSpdY() + 1);
        }
    }

    /**
     * Move the ball according to the calculated x, y, and
rotational
     * velocity.
```

```
     */
    private void updateBallMovement() {
        calculateBallVelX();
        calculateBallVelY();
        calculateBallRotVel();
        // move the ball according to the calculated velocities
        ball.setTranslateY(ball.getTranslateY() + getBallVelY());
        ball.setTranslateX(ball.getTranslateX() + getBallVelX());
        ball.setRotate(ball.getRotate() + getBallRotVel());
    }

    /**
     * Respawns the ball to a random location on the screen along
     * the x-axis fixed at the top 1/3 of the screen on the y-axis.
     *
     */
    private void respawnBallToRandomLocation() {
        // random location in app x-axis (the width of app window)
        ball.setTranslateX(generateRandomBallRespawnXCord());
        ball.setTranslateY(APP_H_Y_CORD_1_3RD_FROM_TOP);
    }

    /**
     * Update the FPS display information based on the current
     * calculations of the FPS, average frame time, and in-game
time.
     *
     */
    private void updateFPSDisplayInformation() {
        // Intermittently update the FPS display information
        if (animationTimerFrameCounter % NUMBER_OF_FRAMES_25 == 0) {
```

```
                fpsDisplayInfoLabel.setText(String.format(
                        FPS_DISPLAY_INFO_STRING,
                        getAvgFPS(),
                        getAvgFpMiliSecond(),
                        getSecondsElapsedInGame()));
            }
        }

        /**
         * A wrapper method that calls all the methods to calculate the
         * in-game time, the average frame time, and the average frames
         * per second.
         *
         * @param now the current time in nanoseconds
         */
        private void setInGameTimeAndAvgFrameTimeAndFPS(long now) {
            setAvgFPS(calculateAvgFPS(now));
            setAvgFpMiliSecond(calculateAvgFpMiliSecond());
            setSecondsElapsedInGame(calculateSecondsElapsedInGame(now));
        }

        /**
         * Generate an x-coordinate for the ball to respawn that is
within
         * the width of the app window
         *
         * from (the left side of the app window) to the width of the
app
         * window minus the width of the ball (the right side of the app
         * window).
         *
         * @return a random x-coordinate for the ball to respawn
```

```
 */
private double generateRandomBallRespawnXCord() {
    return (Math.random() * (BALL_X_CORD_MAX_RESPAWN_LIMIT));
}

/**
 * When the ball hits the bottom of the screen, the game resets.
 *
 */
private void ballHitsBottomOfScreenResetGame() {
    if (ball.getTranslateY() >= APP_H - ball.getHeight()) {
        newGame = true;
    }
}

/**
 * Calculates the average frame time in milliseconds.
 *
 * @return the average frame time in milliseconds
 */
private double calculateAvgFpMiliSecond() {
    // calculate the average frame time in milliseconds (ms)
    return (TIME_01_MILLISECOND / getAvgFPS());
}

/**
 * Calculates the in game time elapsed in seconds.
 *
 * @param now The current time in nanoseconds
 *            (1 billionth of a second)
 * @return the in game time elapsed in seconds
```

```
         */
        private double calculateSecondsElapsedInGame(long now) {
            // convert nanoseconds to seconds
            return (((now - getStartTime()) / TIME_01_SECOND));
        }

        /**
         * calculate the average frame rate per second (FPS)
         *
         * @param now The current time in nanoseconds
         */
        private double calculateAvgFPS(long now) {
            double sum = 0;
            long deltaCurTimePrevTime = now - lastTime;
            // calculate the average FPS
            frameRateArr[animationTimerFrameCounter %
                    frameRateArr.length] = TIME_01_SECOND
                            / deltaCurTimePrevTime;
            lastTime = now;
            // calculate the average frame rate
            for (int i = 0; i < frameRateArr.length; i++) {
                sum += frameRateArr[i];
            }
            return (sum / frameRateArr.length);
        }

    }; // end of game loop
    timer.start(); // Start the game loop
}

public static void main(String[] args) {
```

```
    Application.launch(args);
}

class BinkBonkSound {

    // magic numbers that are not common knowledge unless one
    // has studied the GM2 standard and the midi sound system
    //
    // The initials GM mean General Midi. This GM standard
    // provides for a set of common sounds that respond
    // to midi messages in a common way.
    //
    // MIDI is a standard for the encoding and transmission
    // of musical sound meta-information, e.g., play this
    // note on this instrument at this level and this pitch
    // for this long.
    //
    private static final int MAX_PITCH_BEND = 16383;
    private static final int MIN_PITCH_BEND = 0;
    private static final int REVERB_LEVEL_CONTROLLER = 91;
    private static final int MIN_REVERB_LEVEL = 0;
    private static final int MAX_REVERB_LEVEL = 127;
    private static final int DRUM_MIDI_CHANNEL = 9;
    private static final int CLAVES_NOTE = 76;
    private static final int NORMAL_VELOCITY = 100;
    private static final int MAX_VELOCITY = 127;

    Instrument[] instrument;
    MidiChannel[] midiChannels;
    boolean playSound;
```

```
        public BinkBonkSound() {
            playSound = true;
            try {
                Synthesizer gmSynthesizer = MidiSystem.getSynthesizer();
                gmSynthesizer.open();
                instrument = gmSynthesizer
                        .getDefaultSoundbank()
                        .getInstruments();
                midiChannels = gmSynthesizer.getChannels();

            } catch (MidiUnavailableException e) {
                e.printStackTrace();
            }
        }

        // This method has more comments than would typically be needed for
        // programmers using the Java sound system libraries. This is
because
        // most students will not have exposure to the specifics of midi and
        // the general midi sound system. For example, drums are on channel
        // 10 and this cannot be changed. The GM2 standard defines much of
        // the detail that I have chosen to use static constants to encode.
        //
        // The use of midi to play sounds allows us to avoid using external
        // media, e.g., wav files, to play sounds in the game.
        //
        void play(boolean hiPitch) {
            if (playSound) {

                // Midi pitch bend is required to play a single drum note
                // at different pitches. The high and low pongs are two
```

```
                // octaves apart. As you recall from high school physics,
                // each additional octave doubles the frequency.
                //
                midiChannels[DRUM_MIDI_CHANNEL]
                        .setPitchBend(hiPitch
                                ? MAX_PITCH_BEND
                                : MIN_PITCH_BEND);

                // Turn the reverb send fully off. Drum sounds play until
they
                // decay completely. Reverb extends the audible decay and,
                // from a gameplay point of view, is distracting.
                //
                midiChannels[DRUM_MIDI_CHANNEL]
                        .controlChange(REVERB_LEVEL_CONTROLLER,
                                MIN_REVERB_LEVEL);

                // Play the claves on the drum channel at a "normal" volume
                //
                midiChannels[DRUM_MIDI_CHANNEL]
                        .noteOn(CLAVES_NOTE, NORMAL_VELOCITY);
            }
        }

        public void toggleSound() {
            playSound = !playSound;
        }
    }
}
```

OPongApp.java – PONG 2.0

```
import javafx.animation.AnimationTimer;
import javafx.application.Application;
import javafx.event.EventHandler;
import javafx.scene.control.Label;
import javafx.scene.Cursor;
import javafx.scene.Group;
import javafx.scene.Node;
import javafx.scene.input.KeyCode;
import javafx.scene.input.KeyEvent;
import javafx.scene.input.MouseEvent;
import javafx.scene.paint.Color;
import javafx.scene.Scene;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

import javax.sound.midi.Instrument;
import javax.sound.midi.MidiChannel;
import javax.sound.midi.MidiSystem;
import javax.sound.midi.MidiUnavailableException;
import javax.sound.midi.Synthesizer;

/** sound class, provided in the appendix */
class BinkBonkSound {

    // magic numbers that are not common knowledge unless one
    // has studied the GM2 standard and the midi sound system
    //
```

```
    // The initials GM mean General Midi. This GM standard
    // provides for a set of common sounds that respond
    // to midi messages in a common wySpeed.
    //
    // MIDI is a standard for the encoding and transmission
    // of musical sound meta-information, e.g., plySpeed this
    // note on this instrument at this level and this pitch
    // for this long.
    //
    private static final int MxSpeed_PITCH_BEND = 16383;
    private static final int MIN_PITCH_BEND = 0;
    private static final int REVERB_LEVEL_CONTROLLER = 91;
    private static final int MIN_REVERB_LEVEL = 0;
    private static final int MxSpeed_REVERB_LEVEL = 127;
    private static final int DRUM_MIDI_CHANNEL = 9;
    private static final int CLAVES_NOTE = 76;
    private static final int NORMAL_VELOCITY = 100;
    private static final int MxSpeed_VELOCITY = 127;

    Instrument[] instrument;
    MidiChannel[] midiChannels;
    boolean plySpeedSound;

    public BinkBonkSound() {
        plySpeedSound = true;
        try {
            Synthesizer gmSynthesizer =
MidiSystem.getSynthesizer();
```

```
            gmSynthesizer.open();
            instrument =
gmSynthesizer.getDefaultSoundbank().getInstruments();
            midiChannels = gmSynthesizer.getChannels();

        } catch (MidiUnavailableException e) {
            e.printStackTrace();
        }
    }

    // This method has more comments than would typically be
needed for
    // programmers using the Java sound system libraries. This is
because
    // most students will not have exposure to the specifics of
midi and
    // the general midi sound system. For example, drums are on
channel
    // 10 and this cannot be changed. The GM2 standard defines
much of
    // the detail that I have chosen to use static constants to
encode.
    //
    // The use of midi to plySpeed sounds allows us to avoid
using external
    // media, e.g., wav files, to plySpeed sounds in the game.
    //
    void plySpeed(boolean hiPitch) {
        if (plySpeedSound) {
```

```
            // Midi pitch bend is required to plySpeed a single
drum note
            // at different pitches. The high and low pongs are
two
            // octaves apart. As you recall from high school
physics,
            // each additional octave doubles the frequency.
            //
            midiChannels[DRUM_MIDI_CHANNEL]
                    .setPitchBend(hiPitch ? MxSpeed_PITCH_BEND :
MIN_PITCH_BEND);


            // Turn the reverb send fully off. Drum sounds
plySpeed until they
            // decySpeed completely. Reverb extends the audible
decySpeed and,
            // from a gameplySpeed point of view, is distracting.
            //
            midiChannels[DRUM_MIDI_CHANNEL]
                    .controlChange(REVERB_LEVEL_CONTROLLER,
MIN_REVERB_LEVEL);


            // PlySpeed the claves on the drum channel at a
"normal" volume
            //
            midiChannels[DRUM_MIDI_CHANNEL]
                    .noteOn(CLAVES_NOTE, NORMAL_VELOCITY);
        }
    }


    public void toggleSound() {
```

```
        plySpeedSound = !plySpeedSound;
    }
}

/** class that provides the information displySpeed */
class GameTimer extends Label {

}

/** class that provides the score displySpeed */
class ScoreDisplySpeed extends Label {

}

/**
 * Game object base class
 *
 * A Parent class for all objects used in the game
 * Tracks and models position, velocity, speed
 * Also tracks radius, which allows that to change mid-game
 */
abstract class PhysicsObject {

    // position, velocity, speed
    private double x = 0, y = 0, xSpeed = 0, ySpeed = 0, vx = 0,
vy = 0;

    // constructor
```

```
public PhysicsObject(
        double x,
        double y,
        double xSpeed,
        double ySpeed) {
    this.x = x;
    this.y = y;
    this.xSpeed = xSpeed;
    this.ySpeed = ySpeed;
}

// getters and setters
public double getX() {
    return x;
}

public void setX(double x) {
    this.x = x;
}

public double getY() {
    return y;
}

public void setY(double y) {
    this.y = y;
}
```

```
/**
 * velocity is the rate and direction
 * of an object's movement
 *
 * @param vy
 */
public void setVx(double vx) {
    this.vx = vx;

}

/**
 * velocity is the rate and direction
 * of an object's movement
 *
 * @param vy
 */
public double getVy() {
    return vy;
}

/**
 * velocity is the rate and direction
 * of an object's movement
 *
 * @param vy
 */
public void setVy(double vy) {
```

```
    this.vy = vy;
}

/**
 * velocity is the rate and direction
 * of an object's movement
 *
 * @param vy
 */
public double getVx() {
    return vx;
}

/**
 * Speed is the time rate at which an object
 * is moving along a path
 *
 * @return
 */
public double getSpeedX() {
    return xSpeed;
}

/**
 * Speed is the time rate at which an object
 * is moving along a path
 *
 * @return
```

```
 */
public void setSpeedX(double xSpeed) {
    this.xSpeed = xSpeed;
}

/**
 * Speed is the time rate at which an object
 * is moving along a path
 *
 * @return
 */
public double getSpeedY() {
    return ySpeed;
}

/**
 * Speed is the time rate at which an object
 * is moving along a path
 *
 * @return
 */
public void setSpeedY(double ySpeed) {
    this.ySpeed = ySpeed;
}

// abstract methods to be implemented by subclasses
public abstract void update();
```

```
    /**
     * Velocity (v) is a vector quantity that measures displacement (or change in
     * position) over the change in time (Δt)
     *
     * @return the rate and direction of an object's movement
     */
    public abstract double calculateVx();
}


/**
 * Paddle objects is a rectangle.
 *
 * A paddle is blue
 * A paddle has a width and height
 * A paddle has a position
 * A paddle has a velocity
 * A paddle has a speed
 *
 * The paddle doesn't move on its own, it is controlled by the user's mouse
 * The paddle can only be moved horizontally and it can't move off the screen
 * The paddle is moved by the user's mouse location in the game window
 * The Paddle does not move off the screen
 */
// class Paddle extends PhysicsObject {}
```

```
/**
 * Ball objects is a rectangle that bounces off the top, left, and right bottom
 * screen, bounces off the Paddle object, and falls through the bottom of the
 * screen.
 */
class Ball extends PhysicsObject {


    /**
     * The size of the of "rectangle" that represents the ball this size is
     * used for both the width and the height of the ball
     */
    private static final int BALL_RADIUS = 20;
    // ball is a rectangle
    private static Rectangle ball;
    // Boolean to track if the ball is traveling up or down
    private boolean isBallUp = false;
    // Boolean to track if the ball is traveling left or right
    private boolean isBallLeft = false;


    public static int getBallRadius() {
        return BALL_RADIUS;
    }


    public Rectangle getObject() {
        return ball;
    }
```

```
private boolean isBallUp() {
    return isBallUp;
}

public void setBallUp(boolean isBallUp) {
    this.isBallUp = isBallUp;
}

public boolean isBallLeft() {
    return isBallLeft;
}

public void setBallLeft(boolean isBallLeft) {
    this.isBallLeft = isBallLeft;
}

// constructor
public Ball(
        double x,
        double y,
        double xSpeed,
        double ySpeed) {
    super(x, y, xSpeed, ySpeed);
    ball = new Rectangle(x, y, BALL_RADIUS, BALL_RADIUS);
    ball.setFill(Color.RED);
}
```

```
    @Override
    public void update() {
        setVx(calculateVx());
        setVy(calculateVy());
        ball.setTranslateX(ball.getTranslateX() + getVx());
        ball.setTranslateY(ball.getTranslateY() + getVy());
    }

    @Override
    public double calculateVx() {
        return ((isBallLeft()) ? -getSpeedX() : getSpeedX());
    }

    public double calculateVy() {
        return ((isBallUp()) ? -getSpeedY() : getSpeedY());
    }
}

/**
 * class Pong contains the game model
 * and the game loop it runs in. All the game logic is here.
 *
 * Game logic is the code that determines the state of the
 * game at any given time.
 *
 */
class Pong extends Group {
```

```
    /**
     * Initialize the ball's speed along the y-axis
     */
    private static final int BALL_INITAL_Y_SPEED = 5;

    private static final int _1_3RD_APP_HEIGHT_FROM_TOP =
OPongApp
            .getAppH() / 3;

    // game objects
    // private Paddle paddle;
    private Ball ball = new Ball(
            Math.random() *
                    (OPongApp.getAppW()
                            - Ball.getBallRadius()),
            _1_3RD_APP_HEIGHT_FROM_TOP,
            0,
            BALL_INITAL_Y_SPEED);

    // create game objects
    // paddle = new Paddle(0, 0, 0, 0);

    public Ball getBall() {
        return ball;
    }

    public void startGame() {
        /**
```

```
         * The game loop is a loop that runs until game window is
closed.
         */
        AnimationTimer timer = new AnimationTimer() {

            @Override
            public void handle(long now) {
                // update the game objects
                ball.update();
                if (ball.getObject().getY() > OPongApp
                        .getAppH()- Ball.getBallRadius()) {
                    ball.setBallUp(true);
                }

                if (ball.getObject().getY() < 0) {
                    ball.setBallUp(false);
                }



                System.out.println("ball.getTranslateY() = " +
ball.getObject().getTranslateY());

            }
        };
        timer.start();
    }
}

/**
```

```
 * class PongApp is the main class for the Pong game in MVC architecture
 * this is "the View"
 *
 * This class is the main class for the Pong game.
 * It is responsible for creating the game window and starting the game.
 */
public class OPongApp extends Application {


    /**
     * The title of the game window.
     */
    private static final String APP_TITLE = "OPong (Pong 2.0)";
    /**
     * Pong game window width
     */
    private static final int APP_W = 800;
    /**
     * Pong game window height
     */
    private static final int APP_H = 600;


    /**
     * start method is the main entry point for the JavaFX application
     *
     * The start method allows the application to perform any necessary
```

```
     * initialization before the primary stage is shown
     *
     * @param stage is the game window
     * @throws Exception when the game window can't be created
     */
    @Override
    public void start(Stage primaryStage) throws Exception {

        // create group as root node
        Group root = new Group();

        // scene with root node and size
        Scene scene = new Scene(root, APP_W, APP_H);

        // stage with scene
        primaryStage.setScene(scene);

        // title
        primaryStage.setTitle(APP_TITLE);

        // create game
        Pong pong = new Pong();

        // add game to root node
        root.getChildren().add(pong.getBall().getObject());

        pong.startGame();
```

```
        // show stage
        primaryStage.show();
    }

    public static int getAppW() {
        return APP_W;
    }

    public static int getAppH() {
        return APP_H;
    }
}
```