```
/*********************************************************************************************
 *  This class implements the backend of the Yahtzee Game
 *
 *  It keeps track of the scores and calculates the scoring of the game
 *
 *  CST 283 Programming Assignment2
 *  @author Michael Clinesmith
 *********************************************************************************************/

public class YahtzeeGame
{
    private String[] scoreString = new String[21];
    private int[] scoreLine = new int[21];
    private final boolean[] SCOREABLE_LINES = {false, true, true, true, true, true, true, false, false, false,
            false, true, true, true, true, true, true, false, false, false};
    private boolean[] availableScores= new boolean[21];      // sets up lines that are still available for scoring
    private final int MAX_ROLLS=3, MAX_SCORES=13;
    private int rolls, scoresLeft;
    private int[] diceArray = new int[6];

    /**
     * No-argument constructor, creates the Yahtzee Game
     */
    public YahtzeeGame()
    {
        for(int i=0; i<21; i++)
        {
            scoreString[i]="--";
            scoreLine[i]=0;
            availableScores[i]=SCOREABLE_LINES[i];
        }
        rolls=MAX_ROLLS;
        scoresLeft=MAX_SCORES;
    }

    /**
     * Getter method to get the number of available rolls
     * @return int: The number of rolls remaining in the turn
     */
    public int getRolls()
    {
        return rolls;
    }

    /**
     * Getter method to get the maximum number of rolls in a turn
     * @return int: The maximum number of rolls in a turn
     */
    public int getMAX_ROLLS()
    {
        return MAX_ROLLS;
    }

    /**
     * Setter method for the number of rolls in a turn
     * @param rolls int: The number of rolls to set as remaining
     */
    public void setRolls( int rolls )
    {
        this.rolls = rolls;
```

30/30 points for Program 2

GRADING FOCUS
- GUI (Use of JavaFX; user-friendliness; integration with game class)
- Game class (Require components of any class; general "fit" with the selected game)
- Creativity, originality, and complexity
- Appropriate definition and separation of class data/methods

COMMENTS:
- Way cool solution and above-and-beyond. Easily met basic requirements for assignment.
- Fun to play. Had to stop to keep grading other work.

```
}

/** Method that returns the number of scoring opportunities left in the game
 * @return int: The number of scoresLeft in the game
 */
public int getScoresLeft()
{
    return scoresLeft;
}

/**
 * Method that returns an integer value of a score for the given index
 * @param index int: integer indicating a line on the scorecard
 * @return int: The score of that line
 */
public int getScoreOf(int index)
{
    return scoreLine[index];
}

/**
 * Method that returns the String value of a score for the given index
 * @param index int: integer indicating a line on the scorecard
 * @return String: The String represention of the score
 */
public String getScoreStringOf(int index)
{
    return scoreString[index];
}

/**
 * Method that checks the values for scoring on a certain line
 * @param index int: integer indicating the line to be scored
 * @param dice Die[]: array of dice to be scored
 * @return int: the score for that line given the dice
 */
public int checkScore( int index, Die[] dice)
{
    int score;

    makeDiceArray( dice );      // prepare dice for scoring
    if (index>0 && index<7)
    {
        score = checkScoreUpper(index);
    }
    else if(index>=11 && index<=17)
    {
        score = checkScoreLower(index);
    }
    else
    {
        score = -1;
    }
    return score;
}

/**
 * Method that checks to see what the score in the upper section will be based on the type of dice being scored.
 * @param index int: the index position to check the score on, ranges from 1-6
 * @return int: the score for that index position
```

```
 */
public int checkScoreUpper(int index)
{
    return diceArray[index-1] * index;
}

/**
 * Method that checks to see what the score in the lower section will be based on the type of score being made.
 * It uses the dice that were counted in the DiceArray to make the determination
 * @param index int: the index position to check the score on, ranges from 11-17
 * @return int: the score the would be for that index position
 */
public int checkScoreLower(int index)
{
    int score = 0;

    if (index==11)   // 3 of a kind
    {
        boolean isValid=false;
        for(int i=0; i<6; i++)  // check if 3 of a kind exists
        {
            if (diceArray[i]>=3)
            {
                isValid=true;
            }
        }
        if (isValid)
        {
            score = sumOfDice();
        }
    }
    if (index==12)   // 4 of a kind
    {
        boolean isValid=false;
        for(int i=0; i<6; i++)  // check if 4 of a kind exists
        {
            if (diceArray[i]>=4)
            {
                isValid=true;
            }
        }
        if (isValid)
        {
            score = sumOfDice();
        }
    }
    if (index==13)   // full house - 3 of one number and 2 of another (or possibly 5 of one number)
    {
        boolean has2=false, has3=false, has5=false;

        for(int i=0; i<6; i++)  // check if 4 of a kind exists
        {
            if (diceArray[i]==2)
            {
                has2=true;
            }
            else if (diceArray[i]==3)
            {
                has3=true;
            }
```

```
        else if (diceArray[i]==5)
        {
            has5=true;
        }
    }
    if ((has2 && has3) || has5)
    {
        score = 25;
    }
}
if (index==14)   // Small Straight - must have 1,2,3,4 or 2,3,4,5 or 3,4,5,6
{
    boolean isValid=false;
    if( diceArray[0]>=1 && diceArray[1]>=1 && diceArray[2]>=1 && diceArray[3]>=1)
    {
        isValid=true;
    }
    else if (diceArray[1]>=1 && diceArray[2]>=1 && diceArray[3]>=1 && diceArray[4]>=1)
    {
        isValid=true;
    }
    else if (diceArray[2]>=1 && diceArray[3]>=1 && diceArray[4]>=1 && diceArray[5]>=1)
    {
        isValid=true;
    }

    if (isValid)
    {
        score = 30;
    }
}
if (index==15)  // Large Straight = must have 1,2,3,4,5 or 2,3,4,5,6
{
    boolean isValid=false;
    if (diceArray[1]==1 && diceArray[2]==1 && diceArray[3]==1 && diceArray[4]==1 && (diceArray[0]==1 || diceArray[5]==1))
    {
        isValid = true;
    }
    if (isValid)
    {
        score = 40;
    }
}
if (index==16)   // yahtzee - 5 of one number
{
    boolean has5=false;

    for(int i=0; i<6; i++)  // check if 5 of a kind exists
    {
        if (diceArray[i]==5)
        {
            has5=true;
        }
    }
    if ( has5)
    {
        score = 50;
    }
}
if (index==17)  // chance - just sum 5 dice
```

```
    {
        score = sumOfDice();
    }
    return score;
}

/**
 * Method that sets the score for one of the lines in the scorecard
 * It then also updates the other scores on the scorecard as necessary
 * @param index int: The line on the scorecard to update
 * @param dice Die[]: The array of dice objects used in the scoring
 * @return String: A String representation of the score value that has been set
 */
public String score(int index, Die[] dice)
{
    int value = checkScore( index, dice );
    scoreLine[index]=value;
    scoreString[index] = Integer.toString( value );
    scoreUpdate();

    availableScores[index]=false;               // since scored, this item is no longer available to score
    rolls=MAX_ROLLS;
    scoresLeft--;

    return scoreString[index];
}

/**
 * Method that calculates the sum of the dice counted in the diceArray
 * @return int: The sum of dice in the diceArray
 */
public int sumOfDice()
{
    int sum=0;
    for (int i=0; i<6; i++)
    {
        sum += (i+1) * diceArray[i];    // adds to sum the dice value * number of dice of that value
    }
    return sum;
}

/**
 * This method updates the values that are indirectly changed
 */
public void scoreUpdate()
{
    int upperScore=0, lowerScore=0;

    // check upper section scores
    for (int i=1; i<7; i++) // totals upper score
    {
        upperScore+=scoreLine[i];
    }
    scoreLine[7]= upperScore;
    scoreString[7]=Integer.toString( upperScore );
    if (upperScore>=63)     // check for upper bonus
    {
         scoreLine[8]=35;
         scoreString[8]=Integer.toString( 35 );
         upperScore+=35;
```

```
    }
    scoreLine[9]= upperScore;
    scoreString[9]=Integer.toString( upperScore );
    scoreLine[19]= upperScore;
    scoreString[19]=Integer.toString( upperScore );

    // check lower section scores
    for (int i=11; i<18; i++)
    {
        lowerScore+=scoreLine[i];
    }
    scoreLine[18] = lowerScore;
    scoreString[18]=Integer.toString( lowerScore );
    scoreLine[20] = lowerScore+upperScore;
    scoreString[20]=Integer.toString( scoreLine[20] );
}

/**
 * Method called when a roll is made
 * @return boolean: True if there are rolls remaining, false if not
 */
public boolean madeRoll()
{
    boolean rollsLeft=true;
    rolls--;
    if (rolls<1)
    {
        rollsLeft=false;
    }
    return rollsLeft;
}

/**
 * Method that checks if the games is completed, this is, all scores have been made
 * @return boolean: True if the game is completed, false if not
 */
public boolean isComplete()
{
    boolean complete=true;
    for (int i=0; i<21 && complete; i++)
    {
        if (availableScores[i])     // if a position can still be scored
        {
            complete=false;
        }
    }
    return complete;
}

/**
 * Method to count the number of 1, 2, 3, 4, 5, and 6s on the dice for easier scoring
 * for example:
 * diceArray[0] will store the number of 1s in the Die array
 * ...
 * diceArray[5] will store the number of 6s in the Die array
 *
 * @param dice Die[]: An array of dice with values 1 to 6 to count for each number
 * @return int[]: Not specifically "returned" but calling function can use array to help score the dice
 */
public void makeDiceArray(Die[] dice)
```

```
    {
        int len = dice.length;
        int val;
        //initializes array
        for (int i=0; i<6; i++)
        {
            diceArray[i]=0;
        }
        // counts each number on dice
        for (int i=0; i<len; i++)
        {
            val = dice[i].getValue();   // get value in this Die

            if (val>=1 && val<=6)       // if valid value
            {
                diceArray[val-1]++;
            }
        }
    }

    /**
     * Method to reset the values of the game so a new game can be run
     */
    public void resetGame()
    {
        for(int i=0; i<21; i++)
        {
            scoreString[i]="--";
            scoreLine[i]=0;
            availableScores[i]=SCOREABLE_LINES[i];
        }
        rolls=MAX_ROLLS;
        scoresLeft=MAX_SCORES;
    }

}
```

```
.root {
    -fx-font-family: serif;
    -fx-font-size: 11pt;
    -fx-background-color:  green;
}

.label {
    -fx-text-fill: yellow;
}

.text-area {
    -fx-background-color:  green;
}
```

```
/*********************************************************************************************
 *  This class implements the functionality of a die
 *
 *  Images are loaded for the die images, including blank and questions, and the die can be rolled,
 *  set to certain values and resized
 *
 *  CST 283 Programming Assignment2
 *  @author Michael Clinesmith
 *********************************************************************************************/

import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import java.util.Random;

public class Die
{
    // static fields for loading images - so images don't have to be loaded multiple times
    private static String diceImageName[] = {"Die0.png", "Die1.png", "Die2.png", "Die3.png", "Die4.png",
                                             "Die5.png", "Die6.png", "DieQ.png"};
    private static Image dieImage[] = new Image[8];
    private static ImageView dieImageView[] = new ImageView[8];
    private static boolean imagesLoaded=false;

    private static final int SIDES=6;
    private int value;
    private int size;

    /**
     * No-argument constructor for a die
     * It calls a method to load dice images if they are not yet loaded
     * It will roll the dice to set a random value
     */
    public Die()
    {
        value = 0;
        size = 100;
        if (!imagesLoaded)
        {
            loadImages();
        }
        roll();
    }

    /**
     * Constructor that sets the die to a specific value
     * It calls a method to load dice images if they are not yet loaded
     * @param val int: The value to set the die
     */
    public Die(int val)
    {
        value = val;
        size = 100;
        if (!imagesLoaded)
        {
            loadImages();
        }
    }

    /**
     * Method to load images for the die if they are not yeat loaded
```

```
 * @return boolean: true if the images were successfully loaded, false if not
 */
public boolean loadImages()
{
    int i=0;

    try
    {
        for (i = 0; i < 8; i++)
        {
            dieImage[i] = new Image( "file:" + diceImageName[i] );
            dieImageView[i] = new ImageView( dieImage[i] );
        }
        imagesLoaded = true;
    }
    catch (Exception e)      // problem with loading images
    {
        System.out.println( "Images could not be loaded.\ni="+i );
        System.out.println( e.getClass() );
        imagesLoaded = false;
    }

    return imagesLoaded;
}

/**
 * Method to get an image of the die
 * @return ImageView: The image of the die for the given value
 */
public ImageView getImageView()
{
    ImageView aImageView;
    int val=value;
    if (val<0 || val>6)
    {
        val=7;
    }
    aImageView = new ImageView(dieImage[val]);
    aImageView.setFitWidth( size );
    aImageView.setPreserveRatio( true );
    return aImageView;
}

/**
 * Method to get the value of the die
 * @return int: The value of the die
 */
public int getValue()
{
    return value;
}

/**
 * Method to set the value of the die
 * @param value int: The value the die will be set to
 */
public void setValue( int value )
{
    this.value = value;
}
```

```
    /**
     * Method to get the current setting of the die size
     * @return int: The current set size
     */
    public int getSize()
    {
        return size;
    }

    /**
     * Method to set the size of the displayed die when returning an ImageView
     * @param size int: The size to set an ImageView object to
     */
    public void setSize(int size)
    {
        this.size = size;
    }

    /**
     * Method that rolls the die to a random number from 1 to SIDES
     */
    public void roll()
    {
        Random randomNumbers = new Random();
        value=randomNumbers.nextInt(SIDES)+1;
    }

}
```

```
/*********************************************************************************************
 *  This class contains the main driver of the program and provides a the interface for the Yahtzee Game
 *
 *  The interface includes a Yahtzee scorecard, a title box, 5 dice to be rolled, instructions that can
 *  be viewed, buttons, to roll, and set scores, and to restart and quit the game
 *
 *  It creates a YahtzeeGame object, which it uses to score its lines
 *  It uses Die objects, which are the dice that can be rolled
 *
 *  When the game completes, a popup displays the game score and offers the choice of a new game
 *
 *  CST 283 Programming Assignment2
 *  @author Michael Clinesmith
 *********************************************************************************************/

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

import java.util.Optional;

public class YahtzeeInterface extends Application
{
    private BorderPane mainLayout;
    private YahtzeeGame theGame;        // The game object

    // Scorecard layout objects
    private Label[] scorecardLabels = new Label[21];
    private Label[] scorecardScores = new Label[21];
    private Button[] scorecardButtons = new Button[21];
    private GridPane scorecard;
    private String[] cardLabels = {"Upper Section",
            "Aces = 1",
            "Twos = 2",
            "Threes = 3",
            "Fours = 4",
            "Fives = 5",
            "Sixes = 6",
            "Upper Sum",
            "Bonus for 63+",
            "UPPER TOTAL",
            "Lower Section",
            "3 of a kind",
            "4 of a kind",
            "Full House",
            "Sm. Straight",
            "Lg. Straight",
            "Yahtzee",
            "Chance",
```

```
        "LOWER TOTAL",
        "UPPER TOTAL",
        "COMBINED TOTAL"};
// Denotes lines on scorecard able to directly score
private final boolean[] SCOREABLE_LINES = {false, true, true, true, true, true, true, false, false, false,
        false, true, true, true, true, true, true, true, false, false, false};
private boolean[] availableScores = new boolean[21];      // sets up lines that are still available for scoring

// Dice layout objects
private HBox DiceHolder;
private Die dice[] = new Die[5];
private CheckBox holdDie[] = new CheckBox[5];
private VBox[] diceBox = new VBox[5];

// Action Button layout objects
private Button rollButton;
private Button scoreButton;
private VBox rollBox;

// Top Screen
private Button newGameButton;
private Button quitButton;
private Label yahtzeeTitleLabel;
private HBox TitleBox;

// Informational panels
private RadioButton radioGamePlay, radioScoreExplanation, radioScoreValue;
private ToggleGroup radioGroup;
private VBox radioInfoVBox;
private TextArea gameInstructions, scoringInstructions;
private VBox diceAndInstBox;
private String gameInstruct="Game Play: \n" +
        "Click the roll button to roll the dice\n" +
        "Click the hold button below a die to not\n" +
        "roll that die when the other dice are rolled.\n" +
        "You have up to 3 rolls per turn.\n" +
        "Click the UNLOCK SCORES button to open areas of the \n" +
        "game board to score.\n" +
        "Click the score button to be given an option to score in that position\n" +
        "a popup will show up showing you the value and allowing you\n" +
        "to confirm that score\n" +
        "Cancelling that popup will let you roll again if you have remaining rolls.\n" +
        "After confirming yoru score, roll the dice again for you next turn.\n" +
        "Continue until you complete the game!\n" +
        "Have fun!\n\n";
private String scoringInstruct="Scoring Instructions: \n" +
        "Upper section:\n" +
        "In the upper section, for each line you score only that kind of dice.\n" +
        "For example if you have three 5s and two 3s, \n" +
        "You can score 15 for the 5s, or 6 for the 3s.\n" +
        "Getting at least 63 in the upper section will give\n +" +
        "You a 35 point bonus.\n\n" +
        "Lower section:\n" +
        "For three of a kind-requires at least 3 of one kind of dice to score.\n" +
        "Four of a kind requires at least 4 of one kind of dice,\n" +
        "A full house requires 3 of one kind of dice and 2 of another.\n" +
        "A small straight requires a run of 4 dice, e.g. 2, 3, 4, 5\n" +
        "A large straight requires a run of 5 dice, e.g. 1, 2, 3, 4, 5\n" +
        "A Yahtzee requires 5 of one kind of die , e.g. 4, 4, 4, 4, 4\n" +
        "There is no requirement on the chance.\n\n";
```

```
private String scoringValues= "Scoring values:\n" +
        "Aces: 1 point for each 1\n" +
        "2s: 2 points for each 2\n" +
        "3s: 3 points for each 3\n" +
        "4s: 4 points for each 4\n" +
        "5s: 5 points for each 5\n" +
        "6s: 6 points for each 6\n" +
        "3 of a kind: add all dice\n" +
        "4 of a kind: add all dice\n" +
        "Full House: 25\n" +
        "Small Striaght: 30\n" +
        "Large Straight: 40\n" +
        "Yahtzee: 50\n" +
        "Chance: add all dice\n";

// Roll count layout objects
private Label rollCountLabel;
private Label rollCount;
private VBox rollCountBox;

/**
 * Starting method of application - calls launch
 * @param args String[]: Not used
 */
public static void main( String[] args )
{
    // Launch the application.
    launch( args );
}

/**
 * Method that calls the initializeScene method and creates the scene
 *
 * @param primaryStage Stage object used to create the stage
 */
@Override
public void start( Stage primaryStage )
{
    initializeScene();

    // Set up overall scene
    Scene scene = new Scene( mainLayout, 1100, 900 );
    scene.getStylesheets().add( "Yaht.css" );
    primaryStage.setScene( scene );
    primaryStage.setTitle( "YAHTZEE! - Game design by Michael Clinesmith" );
    primaryStage.show();
}

/**
 * Method that calls other methods to create the game, create the interface, then sets up the mainLayout
 */
public void initializeScene()
{
    theGame = new YahtzeeGame();
    createTitle();
    createInfoBoxes();
    createScorecard();
    createRollButtons();
    createDiceHolders();
```

```
    mainLayout = new BorderPane();
    mainLayout.setTop( TitleBox );
    mainLayout.setLeft( scorecard );
    mainLayout.setCenter( diceAndInstBox );
}

/**
 * Method that creates the interface parts at the top of the screen, the name and new game and quit buttons
 */
public void createTitle()
{
    newGameButton = new Button( "NEW GAME" );
    newGameButton.setOnAction( new GameButtonHandler() );

    quitButton = new Button( "QUIT" );
    quitButton.setOnAction( new GameButtonHandler() );

    yahtzeeTitleLabel = new Label( "YAHTZEE!" );
    yahtzeeTitleLabel.setStyle( "-fx-font-size: 24; -fx-text-fill:  orange" );

    TitleBox = new HBox( 50, yahtzeeTitleLabel, newGameButton, quitButton );
    TitleBox.setAlignment( Pos.CENTER );
    TitleBox.setPadding( new Insets( 20 ) );
}

/**
 * Method that sets up the information box in the center of the interface and the radio
 * buttons that control the information in the box
 */
public void createInfoBoxes()
{
    // create radiobuttons that control info in text area
    radioGamePlay = new RadioButton( "Game Play" );
    radioScoreExplanation = new RadioButton( "Score Explanation" );
    radioScoreValue = new RadioButton( "Score Values" );

    radioGroup = new ToggleGroup();
    radioGamePlay.setToggleGroup( radioGroup );
    radioGamePlay.setSelected( true );
    radioGamePlay.setOnAction( new RadioHandler() );
    radioScoreExplanation.setToggleGroup( radioGroup );
    radioScoreExplanation.setOnAction( new RadioHandler() );
    radioScoreValue.setToggleGroup( radioGroup );
    radioScoreValue.setOnAction( new RadioHandler() );

    radioInfoVBox= new VBox( 10, radioGamePlay, radioScoreExplanation, radioScoreValue  );
    radioInfoVBox.setPadding( new Insets( 10 ) );
    radioInfoVBox.setAlignment( Pos.CENTER );

    // create textArea
    gameInstructions = new TextArea( gameInstruct);
    gameInstructions.setEditable( false );
    gameInstructions.setPadding( new Insets( 50 ) );
    gameInstructions.setPrefRowCount( 25 );

}

/**
 * Method that creates the buttons next to the dice for rolling and scoring
 */
```

```
public void createRollButtons()
{
    rollButton = new Button( "ROLL DICE" );
    rollButton.setOnAction( new RollButtonHandler() );

    scoreButton = new Button( "UNLOCK SCORES" );
    scoreButton.setOnAction( new RollButtonHandler() );
    scoreButton.setDisable( true );     // score Button initially disabled

    rollBox = new VBox( 10, rollButton, scoreButton );
    rollBox.setAlignment( Pos.CENTER );
}

/**
 * Method that creates the Yahtzee Scorecard
 * It also sets the values in the associated availableScores Array controlling the scoreable fields
 */
public void createScorecard()
{
    scorecard = new GridPane();
    for (int i = 0; i < 21; i++)
    {
        scorecardLabels[i] = new Label( cardLabels[i] );
        scorecardLabels[i].setPadding( new Insets( 10 ) );
        scorecardScores[i] = new Label( "--" );
        scorecardScores[i].setPadding( new Insets( 10 ) );
        if (SCOREABLE_LINES[i])         // if this line is directly scoreable
        {
            scorecardButtons[i] = new Button( "score" );
        } else
        {
            scorecardButtons[i] = new Button( "-----" );
        }

        scorecardButtons[i].setOnAction( new ScoringButtonHandler() );
        scorecardButtons[i].setDisable( true );
        scorecard.add( scorecardLabels[i], 0, i );
        scorecard.add( scorecardScores[i], 1, i );
        scorecard.add( scorecardButtons[i], 2, i );
        scorecard.setAlignment( Pos.CENTER );
        scorecard.setGridLinesVisible( true );
        availableScores[i] = SCOREABLE_LINES[i];        // sets the lines on the scorecard that can be scored
    }

    // highlight important sections
    scorecardScores[9].setStyle( "-fx-font-weight: bold" );     // Upper Total
    scorecardLabels[9].setStyle( "-fx-font-weight: bold" );
    scorecardScores[18].setStyle( "-fx-font-weight: bold" );    // Lower Total
    scorecardLabels[18].setStyle( "-fx-font-weight: bold" );
    scorecardScores[20].setStyle( "-fx-font-weight: bold" );    // Combined Total
    scorecardLabels[20].setStyle( "-fx-font-weight: bold" );

    scorecard.setPadding( new Insets( 10 ) );
}

/**
 * Method that creates the Dice, the hold checkboxes, the rolls remaining fields
 * and combines them along with the roll buttons, and information controls into the diceAndInstBox
 */
public void createDiceHolders()
```

```
{
    for (int i = 0; i < 5; i++)
    {
        dice[i] = new Die( 7 );
        dice[i].setSize( 80 );
        holdDie[i] = new CheckBox( "Hold" );
        holdDie[i].setPadding( new Insets( 10 ) );
        holdDie[i].setDisable( true );      // initially disable hold when invalid dice
        diceBox[i] = new VBox( 10, dice[i].getImageView(), holdDie[i] );
        diceBox[i].setAlignment( Pos.CENTER );
    }

    // labels for keeping track of rolls remaining
    rollCountLabel = new Label( "Rolls Remaining:" );
    rollCountLabel.setAlignment( Pos.CENTER );
    rollCount = new Label( Integer.toString( theGame.getRolls() ) );
    rollCount.setAlignment( Pos.CENTER );
    rollCountBox = new VBox( 10, rollCountLabel, rollCount );
    rollCountBox.setAlignment( Pos.CENTER );

    // Put together Dice Holder box
    DiceHolder = new HBox( 10, diceBox );
    DiceHolder.getChildren().add( rollBox );
    DiceHolder.getChildren().add( rollCountBox );
    DiceHolder.setAlignment( Pos.CENTER );
    DiceHolder.setPadding( new Insets(  50) );
    // puts dice, radiobuttons controlling info, and text area together
    diceAndInstBox = new VBox( 10,  DiceHolder , radioInfoVBox, gameInstructions);
}

/**
 * Method that unlocks the scoring buttons that have not yet been scored so the user can select them
 */
public void unlockScoringButtons()
{
    for (int i = 0; i < 21; i++)
    {
        if (availableScores[i])
        {
            scorecardButtons[i].setDisable( false );
        }
    }
    rollButton.setDisable( true );  // locks rolling button while scoring
}

/**
 * Method that locks the scoring buttons preventing scoring when not applicable
 */
public void lockScoringButtons()
{
    for (int i = 0; i < 21; i++)
    {
        scorecardButtons[i].setDisable( true );
    }
}

/**
 * Method that prevents the hold die locks from being selected when not applicable
 */
public void lockHolds()
```

```
{
    for (int i = 0; i < 5; i++)
    {
        holdDie[i].setDisable( true );
    }
}

/**
 * Methood that prepares for the next round after scoring or a new game by locking the scoring buttons, resetting the dice,
 * locking the dice holds, locking the score button, unlocking the roll button and resetting the rolls
 */
public void prepareNextRound()
{

    lockScoringButtons();

    // resets dice
    for (int i = 0; i < 5; i++)
    {
        dice[i].setValue( 7 );
        diceBox[i].getChildren().clear();
        diceBox[i].getChildren().addAll( dice[i].getImageView(), holdDie[i] );
        holdDie[i].setSelected( false );
        diceBox[i].setAlignment( Pos.CENTER );
    }
    lockHolds();

    // locks score button by dice
    scoreButton.setDisable( true );
    // unlocks roll button by dice
    rollButton.setDisable( false );

    rollCount.setText( Integer.toString( theGame.getRolls() ) );    // set roll count label

}

/**
 * Method that updates the fields on the scorecard
 */
public void updateScores()
{
    for (int i = 0; i < 21; i++)
    {
        scorecardScores[i].setText( theGame.getScoreStringOf( i ) );
    }

}

/**
 * Method that runs after a game has been completed, displaying the score and
 * giving the user the option to restart the game
 */
public void gameCompleted()
{
    lockScoringButtons();
    String message = "You completed the game!  Your final score was " + scorecardScores[20].getText() + "!" +
            "\nClick OK to reset the game board and play again!";

    Alert alert = new Alert( Alert.AlertType.CONFIRMATION );
    alert.setTitle( "Final Score" );
```

```
    alert.setContentText( message );
    Optional<ButtonType> result = alert.showAndWait();
    if (result.get() == ButtonType.OK)
    {
        resetGame();
    }

}

/**
 * Method that resets the game
 * It resets the scores by calling the YahtzeeGame object, updates those scores to the interface,
 * the prepares for the next round
 */
public void resetGame()
{
    theGame.resetGame();
    updateScores();
    prepareNextRound();
    resetAvailableScores();
}

/**
 * Method that resets the availableScores array for a new game!
 */
public void resetAvailableScores()
{
    for (int i=0; i<21; i++)
    {
        availableScores[i] = SCOREABLE_LINES[i];        // sets the lines on the scorecard that can be scored
    }
}

/**
 * Class that handles the ActionEvents from the roll and unlock scoring buttons
 */
class RollButtonHandler implements EventHandler<ActionEvent>
{
    /**
     * Method that handles the roll and unlock scoring button clicks
     * @param event ActionEvent: event triggered by a button click
     */
    @Override
    public void handle( ActionEvent event )
    {
        boolean stillRolls = true;
        if (event.getSource() == rollButton)      // dice are rolled
        {
            for (int i = 0; i < 5; i++)
            {
                if (holdDie[i].isSelected() == false)
                {
                    dice[i].roll();
                    // clear and read dice back to the diceBox with new images
                    diceBox[i].getChildren().clear();
                    diceBox[i].getChildren().addAll( dice[i].getImageView(), holdDie[i] );
                    diceBox[i].setAlignment( Pos.CENTER );
                }
                holdDie[i].setDisable( false ); // unlock holds on roll
            }
```

```
            scoreButton.setDisable( false );    // possible to score

            // make updates on roll number
            stillRolls = theGame.madeRoll();                                    // check if still rolls available
            rollCount.setText( Integer.toString( theGame.getRolls() ) );    // set roll count label
            if (!stillRolls)                                                    // disable rolls if no rolls remaining
            {
                rollButton.setDisable( true );
            }
        }
        if (event.getSource() == scoreButton)     // prepare to score
        {
            unlockScoringButtons();
        }

    }
}

/**
 * Class that handles the ActionEvents for the scoring buttons on the Yahtzee scorecard
 */
class ScoringButtonHandler implements EventHandler<ActionEvent>
{
    /**
     * Method that handles the ActionEvents for the scoring buttons on the Yahtzee scorecard
     * @param event ActionEvent: event triggered by a button click
     */
    @Override
    public void handle( ActionEvent event )
    {
        String message = "";
        int score = 0;
        String scoreString = "";

        for (int i = 0; i < 21; i++)
        {
            if (event.getSource() == scorecardButtons[i])
            {
                score = theGame.checkScore( i, dice );
                message = "The scoring value for " + scorecardLabels[i].getText() + " is " + score + "." +
                        "\n Do you want to make this score?";

                Alert alert = new Alert( Alert.AlertType.CONFIRMATION );
                alert.setTitle( "Verify Score" );
                alert.setContentText( message );
                Optional<ButtonType> result = alert.showAndWait();
                if (result.get() == ButtonType.OK)
                {
                    scoreString = theGame.score( i, dice );
                    updateScores();
                    scorecardScores[i].setText( scoreString );
                    availableScores[i] = false;
                    if (theGame.isComplete())
                    {
                        gameCompleted();
                    } else
                    {
                        prepareNextRound();
                    }
                } else    // decide not to score, so lock scoring buttons and unlock roll if still rolls
```

```
                {
                    lockScoringButtons();
                    if (theGame.getRolls() > 0)
                    {
                        rollButton.setDisable( false );
                    }
                }
            }
        }

}

/**
 * Class that handles ActionEvents for setting a new game or quitting
 */
class GameButtonHandler implements EventHandler<ActionEvent>
{
    /**
     * Method that handles ActionEvents for the new game and quit buttons
     * @param event ActionEvent: Event caused by clicking the new game and quit buttons
     */
    @Override
    public void handle( ActionEvent event )
    {
        String message;

        if (event.getSource() == newGameButton)      // user chooses new game
        {
            message = "Do you want to restart the game?";

            Alert alert = new Alert( Alert.AlertType.CONFIRMATION );
            alert.setTitle( "Restart" );
            alert.setContentText( message );
            Optional<ButtonType> result = alert.showAndWait();
            if (result.get() == ButtonType.OK)
            {
                resetGame();
            }
        }
        else if (event.getSource() == quitButton)   // user chooses to quit
        {
            message = "Do you want to quit the game?";

            Alert alert = new Alert( Alert.AlertType.CONFIRMATION );
            alert.setTitle( "Quit?" );
            alert.setContentText( message );
            Optional<ButtonType> result = alert.showAndWait();
            if (result.get() == ButtonType.OK)
            {
                System.exit( 0 );
            }
        }
    }
}

/**
 * Class that handles ActionEvents by clicking on the information radiobuttons
 */
class RadioHandler implements EventHandler<ActionEvent>
```

```
    {
        /**
         * Method that handles the ActionEvents of clicking on the information radiobuttons
         * @param event ActionEvent: event created by clicking on a radiobutton
         */
        @Override
        public void handle( ActionEvent event )
        {
            // set information in textarea based on user choice
            if (event.getSource() == radioGamePlay)
            {
                gameInstructions.setText( gameInstruct );

            } else if (event.getSource() == radioScoreExplanation)
            {
                gameInstructions.setText( scoringInstruct );


            } else if (event.getSource() == radioScoreValue)
            {
                gameInstructions.setText( scoringValues );
            }
        }
    }
}
```