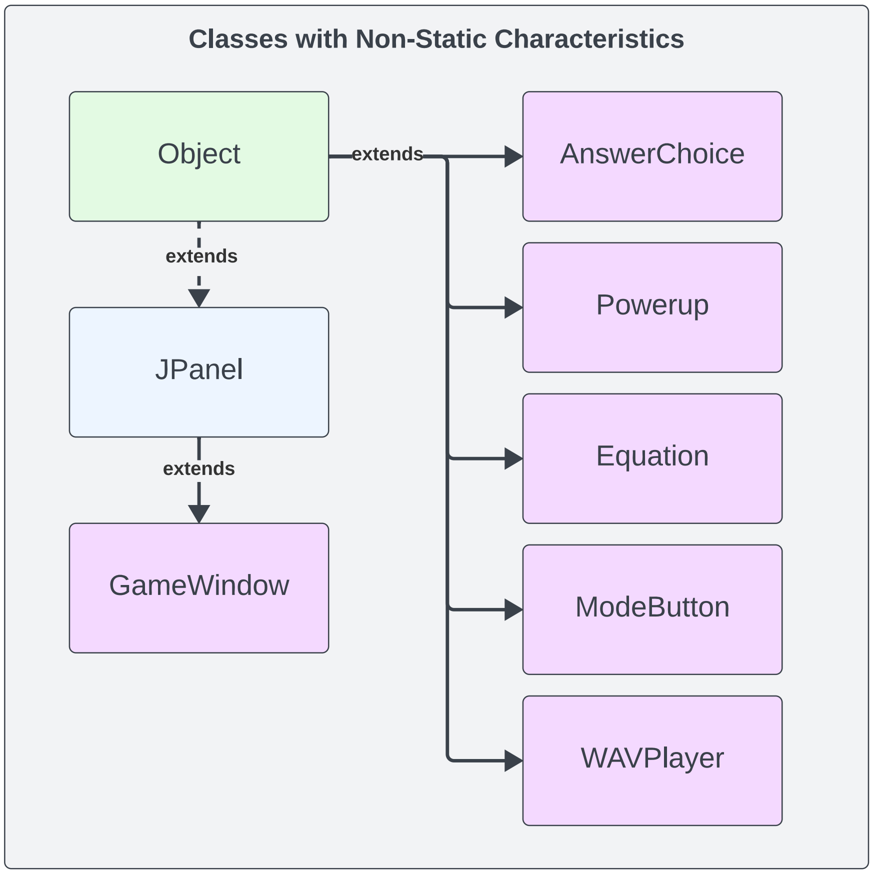
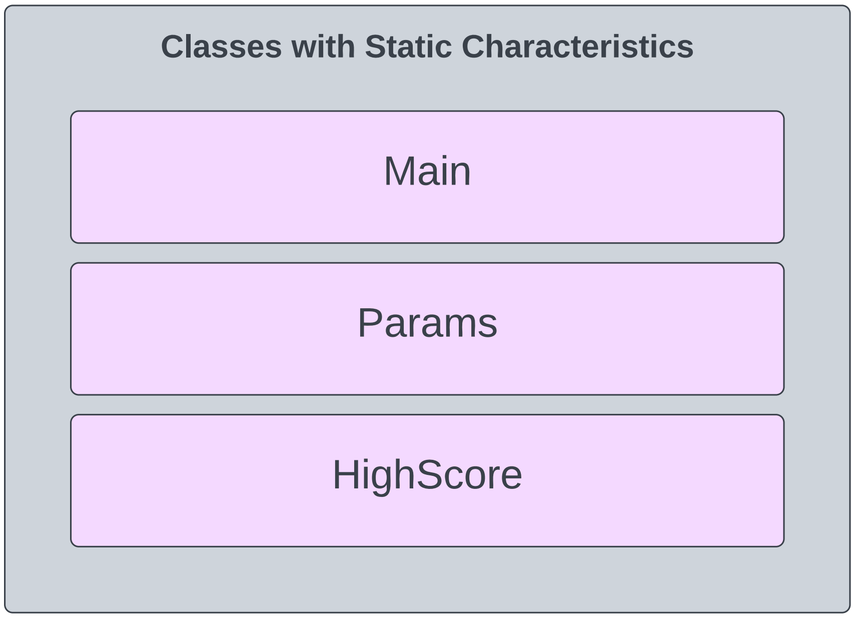
COMPUCLICK Program Specifications

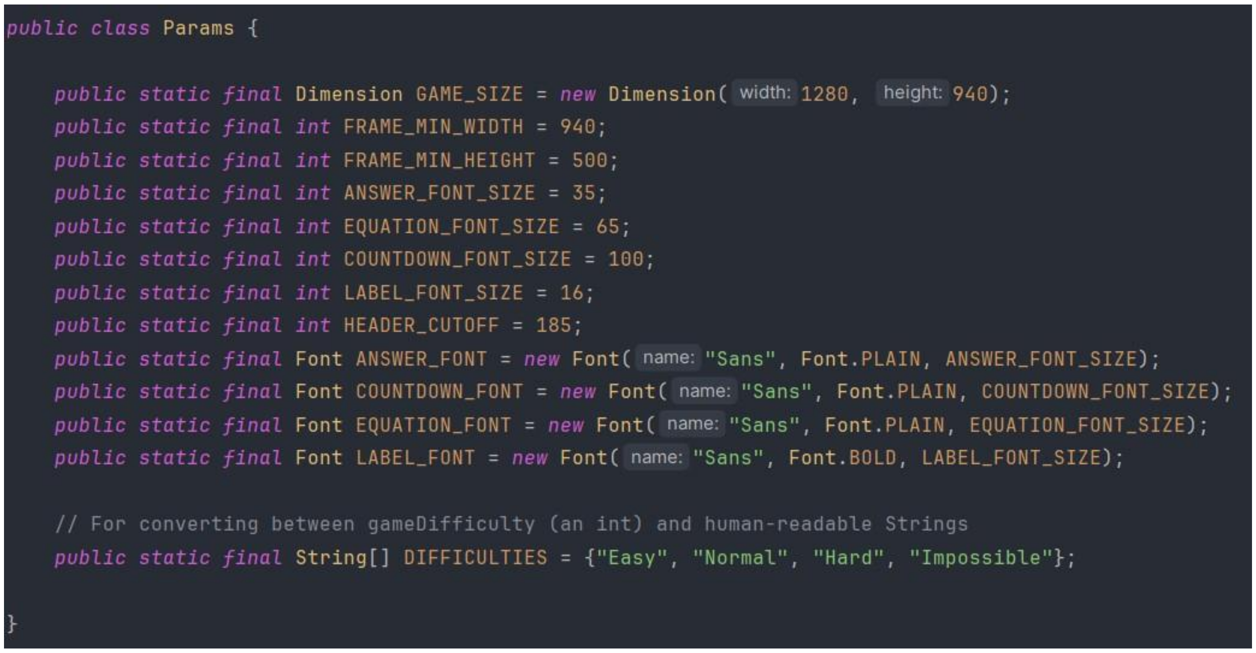




1. Program Overview

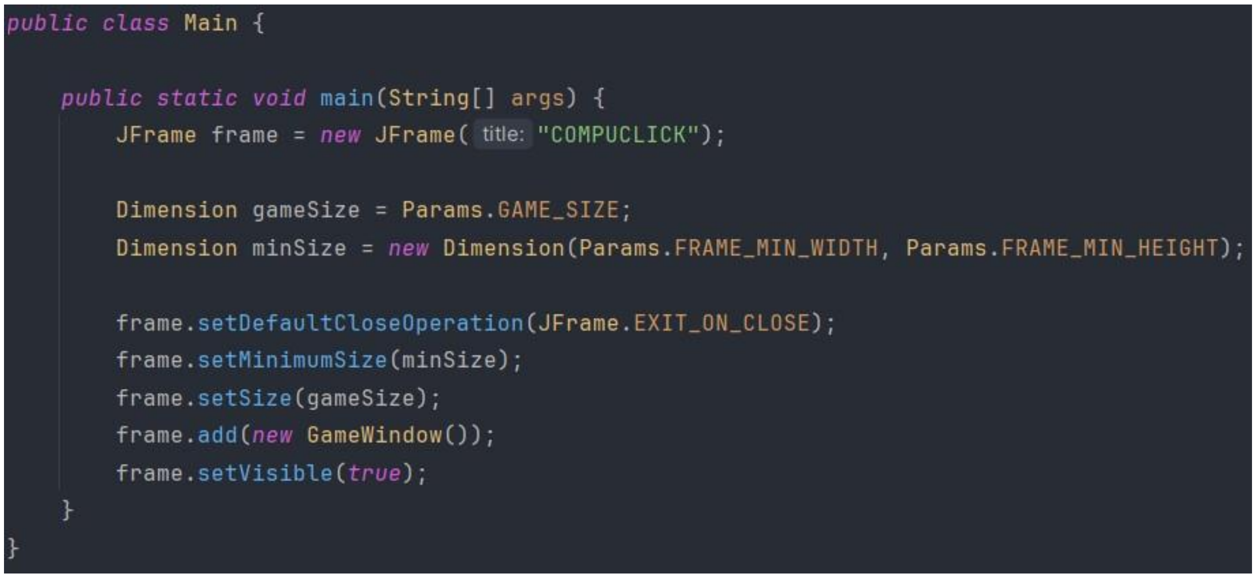
COMPUCLICK is composed of a total of nine custom-written Java classes, six of which are instantiated and used as Objects, and three of which simply contain static methods and state.

The simplest class is **Params**, which contains static variables that dictate certain visual elements of various components of the game:



```java
public class Params {

    public static final Dimension GAME_SIZE = new Dimension( width: 1280,  height: 940);
    public static final int FRAME_MIN_WIDTH = 940;
    public static final int FRAME_MIN_HEIGHT = 500;
    public static final int ANSWER_FONT_SIZE = 35;
    public static final int EQUATION_FONT_SIZE = 65;
    public static final int COUNTDOWN_FONT_SIZE = 100;
    public static final int LABEL_FONT_SIZE = 16;
    public static final int HEADER_CUTOFF = 185;
    public static final Font ANSWER_FONT = new Font( name: "Sans", Font.PLAIN, ANSWER_FONT_SIZE);
    public static final Font COUNTDOWN_FONT = new Font( name: "Sans", Font.PLAIN, COUNTDOWN_FONT_SIZE);
    public static final Font EQUATION_FONT = new Font( name: "Sans", Font.PLAIN, EQUATION_FONT_SIZE);
    public static final Font LABEL_FONT = new Font( name: "Sans", Font.BOLD, LABEL_FONT_SIZE);

    // For converting between gameDifficulty (an int) and human-readable Strings
    public static final String[] DIFFICULTIES = {"Easy", "Normal", "Hard", "Impossible"};

}
```

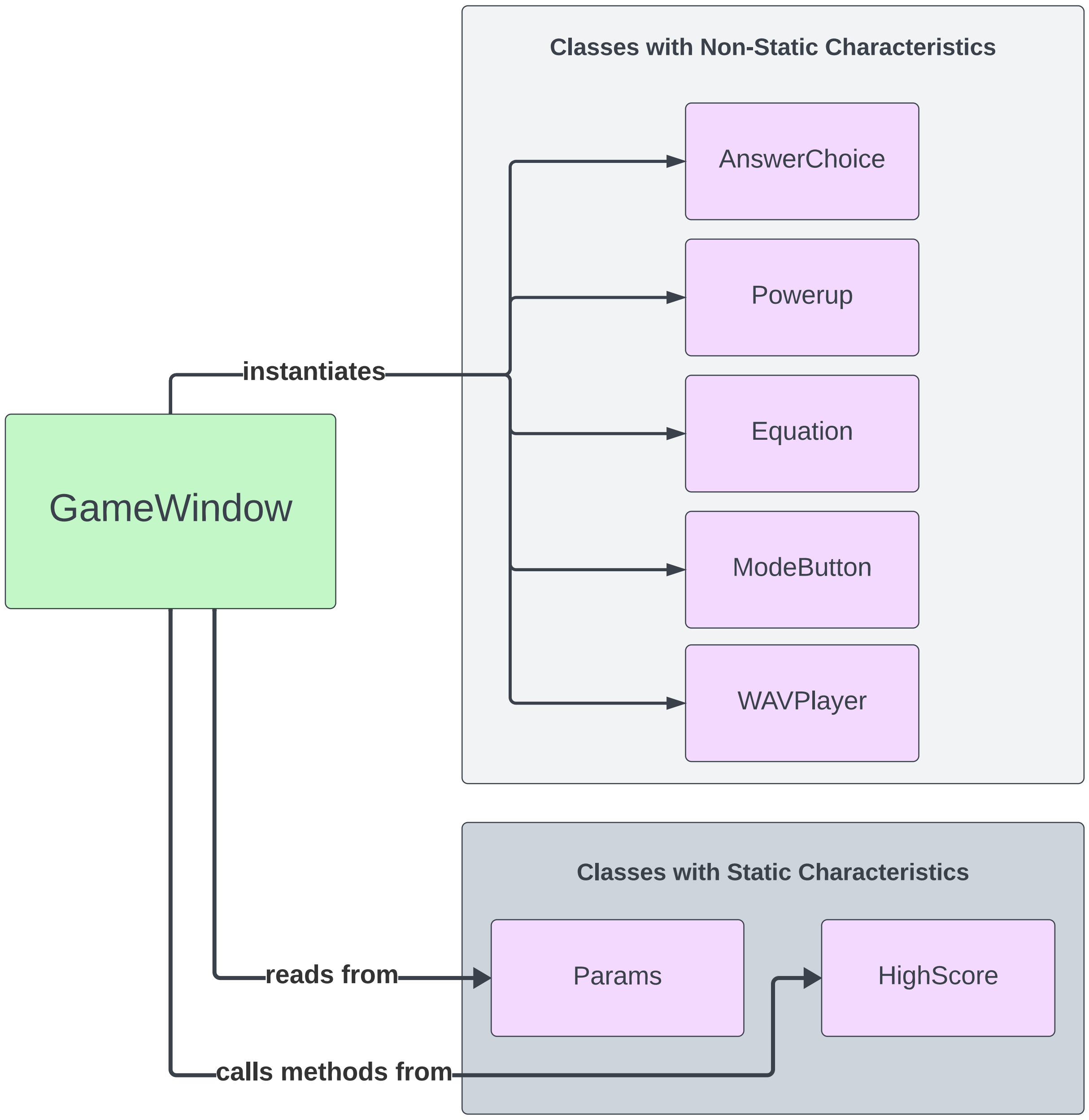
The **Main** class is also quite simple, as it contains the code for creating the JFrame and instantiating the JPanel in which the game runs:



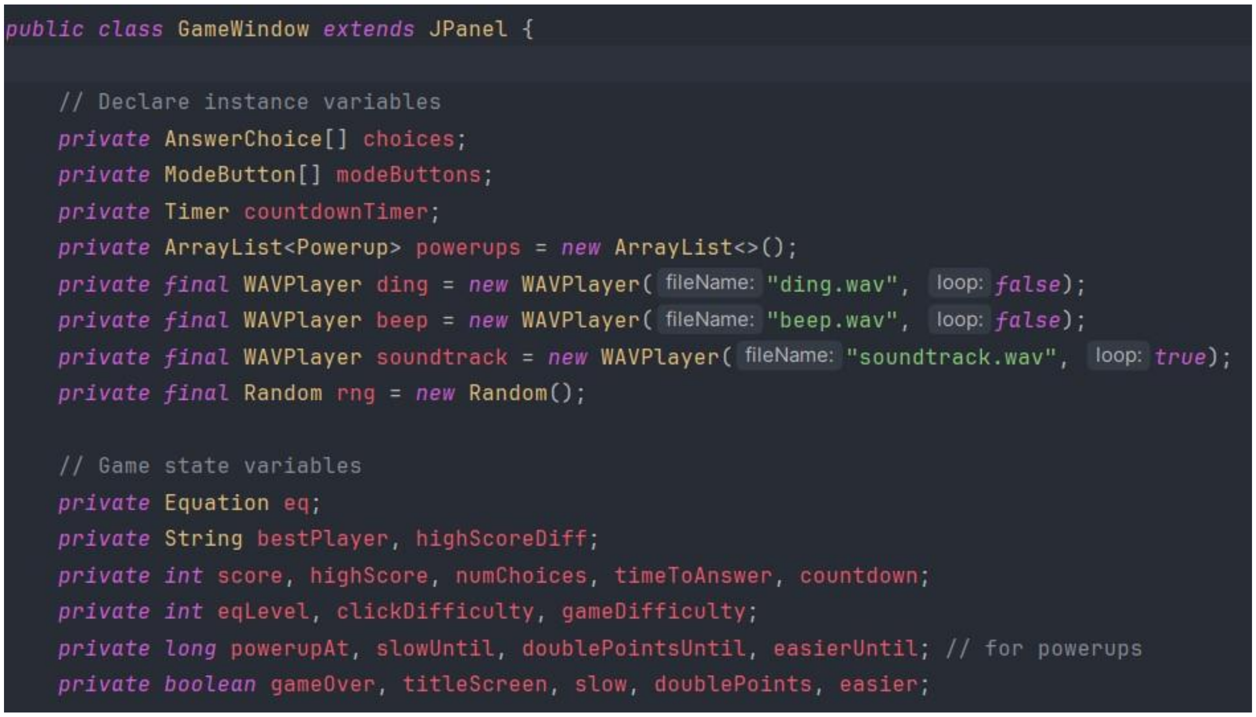
```java
public class Main {

    public static void main(String[] args) {
        JFrame frame = new JFrame( title: "COMPUCLICK");

        Dimension gameSize = Params.GAME_SIZE;
        Dimension minSize = new Dimension(Params.FRAME_MIN_WIDTH, Params.FRAME_MIN_HEIGHT);

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setMinimumSize(minSize);
        frame.setSize(gameSize);
        frame.add(new GameWindow());
        frame.setVisible(true);
    }
}
```

2. The GameWindow Class

As we see from **Main**, we instantiate a **GameWindow** object (a child of JPanel) to add to the JFrame. **GameWindow** is where the bulk of the game logic is stored, so, to increase readability, here is a diagram that depicts how **GameWindow** interacts with the other classes:



GameWindow uses several private instance variables, all of which are important in keeping track of the game state:



```
public class GameWindow extends JPanel {

    // Declare instance variables
    private AnswerChoice[] choices;
    private ModeButton[] modeButtons;
    private Timer countdownTimer;
    private ArrayList<Powerup> powerups = new ArrayList<>();
    private final WAVPlayer ding = new WAVPlayer( fileName: "ding.wav",  loop: false);
    private final WAVPlayer beep = new WAVPlayer( fileName: "beep.wav",  loop: false);
    private final WAVPlayer soundtrack = new WAVPlayer( fileName: "soundtrack.wav",  loop: true);
    private final Random rng = new Random();

    // Game state variables
    private Equation eq;
    private String bestPlayer, highScoreDiff;
    private int score, highScore, numChoices, timeToAnswer, countdown;
    private int eqLevel, clickDifficulty, gameDifficulty;
    private long powerupAt, slowUntil, doublePointsUntil, easierUntil; // for powerups
    private boolean gameOver, titleScreen, slow, doublePoints, easier;
```

First, we have arrays for storing the available answer choices, and for storing the title screen difficulty (mode) buttons. **Powerup** objects are stored in an ArrayList, as they need to be added and removed on the fly. We also instantiate three **WAVPlayer** objects, which respectively allow the game to play the soundtrack, a ding sound for correct clicks, and a beep sound for incorrect clicks. We also instantiate a Timer with which to decrement the countdown variable.

The primary game state variables include (in order of appearance):

- **`eq`**: The current round's equation.
- **`bestPlayer`**: the current champion ("best player").
- **`highScoreDiff`**: the difficulty the champion played on.
- **`score`**: the current player's score.
- **`highScore`**: the current champion's high score.
- **`numChoices`**: the number of choices to be drawn.
- **`timeToAnswer`**: the amount of time the player has in total to answer in the round.
- **`countdown`**: the countdown in 10s of milliseconds (which decreases during the round).
- **`eqLevel`**: the equation difficulty.
- **`clickDifficulty`**: the click difficulty (how hard it is to click the answer choices).
- **`gameDifficulty`**: the current game difficulty set by the player.
- **`powerupAt`**: the time in milliseconds since 1970 at which to add a powerup.
- **`slowUntil`**: the time in milliseconds since 1970 until which to slow time.

- **`doublePointsUntil`**: the time in milliseconds since 1970 until which to give the player twice as many points per successful click.
- **`easierUntil`**: the time in milliseconds since 1970 until which to make the equations one level easier.
- **`gameOver`**: whether or not the game is over.
- **`titleScreen`**: whether or not to show the title screen.
- **`slow`**: whether or not time is currently slowed.
- **`doublePoints`**: whether or not points are currently being doubled.
- **`easier`**: whether or not the equations are currently being made easier.

The **GameWindow** class can be divided into four major sections:

1. Declaration of instance variables (just discussed).
2. Declaration of methods that modify game state.
3. Drawing the game state using Swing and Graphics.
4. Event handling using inner classes.

With respect to **section 2**, we have four methods of significance:

- **`private void resetGame()`**: starts a new game by setting all necessary instance variables back to their initial conditions. **`gameDifficulty`** is already set by the player's clicking a title screen button by this point, and so **`resetGame()`** doesn't modify it. Calls **`resetChoices()`** during its control flow.
- **`private void resetChoices()`**: populates the **`choices`** array with both the current equation's correct answer, and some number of incorrect answers.
- **`private void nextRound()`**: advances the game to the next round by generating a new equation, adding a point to the score, and resetting the countdown. This method is called when the player clicks on the correct answer choice.
- **`private void gameOver()`**: freezes the game by removing all powerups, stopping the countdown timer, and halting the random movements of each answer choice. It also plays an unpleasant sound to alert the user, and sets **`gameOver`** to true (so that paintComponent() knows to alert the user via a visual message).

With respect to **section 3**, we have three methods of significance:

- **`public void paintComponent(Graphics g)`**: overrides the inherited JPanel paintComponent() method, and facilitates drawing everything the player sees. This method also calls the two subsequent methods to reduce code complexity.
- **`private void drawGameUI(Graphics g)`**: responsible for drawing the top portion of the screen, including the current equation, countdown, score, and high score information.

- **`private void drawTitleScreen(Graphics g)`**: responsible for drawing the title screen (difficulty selector) portion of the UI. This method is only called after a game ends or after the user launches the program.

With respect to **section 4**, we have four inner classes of significance:

- **`private class MouseHandler`**: extends the MouseAdapter class and, specifically, overrides the mousePressed() method. This class performs the following functions:
  - Determine if the mouse was pressed within a title screen mode selection button by calling **`ModeButton.contains()`** and adjust the game state accordingly if so.
  - Determine if the mouse was pressed within the hitbox of a powerup by calling **`Powerup.contains()`** and adjust the game state accordingly if so.
  - Determine if the mouse was pressed within the hitbox of an answer choice by calling **`AnswerChoice.contains()`** and adjust the game state accordingly if so, depending on whether the answer choice was correct or not. If difficulty is set to 3 ("Impossible"), then this will call **`gameOver()`** if the user presses *anything but* the correct answer (including the background).
- **`private class KeyHandler`**: extends the KeyAdapter class and, specifically, overrides the keyPressed() method. The sole purpose of this class is to detect, whenever a game ends, if the user presses the **R** key, which will bring them back to the mode selector / title screen.
- **`private class CountdownTimerHandler`**: implements the ActionListener interface and therefore the actionPerformed() method. This class is also quite simple, as its purpose is to decrement the **`countdown`** instance variable by 10 each time the Timer it serves fires. When **`countdown`** reaches 0, **`gameOver()`** is called.
- **`private class RefreshTimerHandler`**: implements the ActionListener interface and therefore the actionPerformed() method. The class serves as the ActionListener for the game's **`refreshTimer`**, which fires every 15 milliseconds, and which facilitates the animation of various visual elements. **`refreshTimer`** is instantiated in the **`GameWindow`** constructor, and is active as long as the program remains open. This class performs the following functions:
  - Call repaint() to facilitate animation within the JPanel.
  - Randomize the velocities of each **`AnswerChoice`** in the **`choices`** array based on the **`clickDifficulty`** instance variable.

- Call **`AnswerChoice.moveAnswer()`** for each **`AnswerChoice`** in the **`choices`** array to update their positions on the screen.

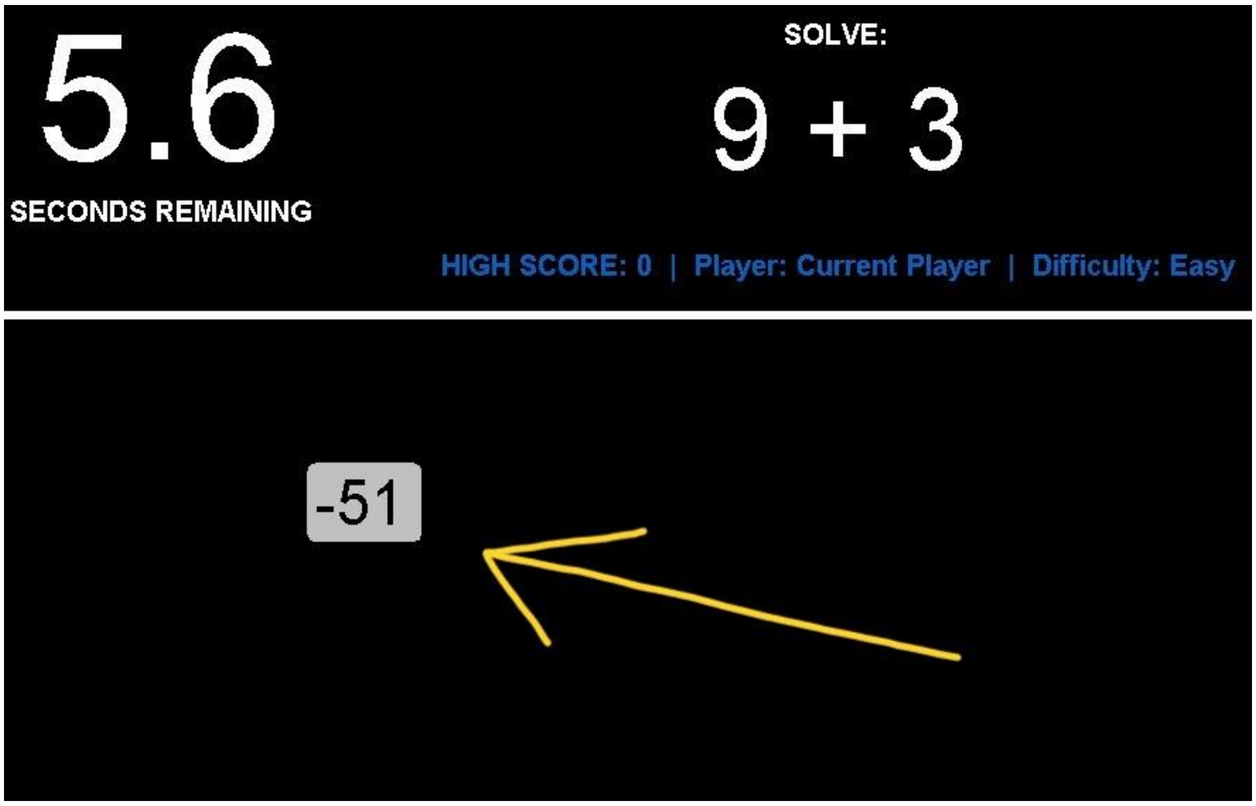
- Prevent answer choices from moving off screen when window is resized / as they move about the screen by calling **`AnswerChoice.checkOutOfBounds()`** for each **`AnswerChoice`** in the **`choices`** array.

- Add a **`Powerup`** every 15 to 30 seconds to the **`powerups`** array depending on **`powerupAt`**.

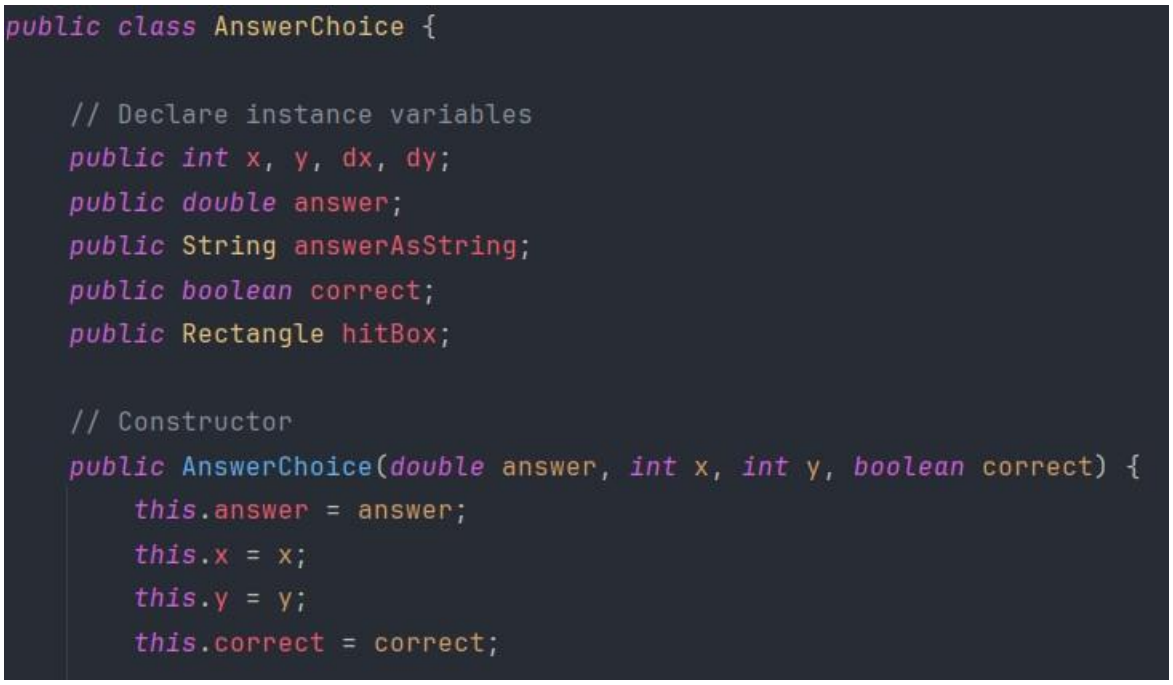
- Prevent powerups from moving off screen when window is resized by calling **`Powerup.checkOutOfBounds()`** for each **`Powerup`** in the **`powerups`** array.

- Remove powerups when they expire after 10 seconds based on **`slowUntil, doublePointsUntil, and easierUntil`**, adjusting game state variables as necessary.

3. The AnswerChoice Class



The **`AnswerChoice`** class stores information about the answer choices that appear on the screen during each game round. **`AnswerChoice`** instances have the following variables, and take the following constructor parameters:



```
public class AnswerChoice {

    // Declare instance variables
    public int x, y, dx, dy;
    public double answer;
    public String answerAsString;
    public boolean correct;
    public Rectangle hitBox;

    // Constructor
    public AnswerChoice(double answer, int x, int y, boolean correct) {
        this.answer = answer;
        this.x = x;
        this.y = y;
        this.correct = correct;
```

<u>The most significant instance variables are as follows:</u>

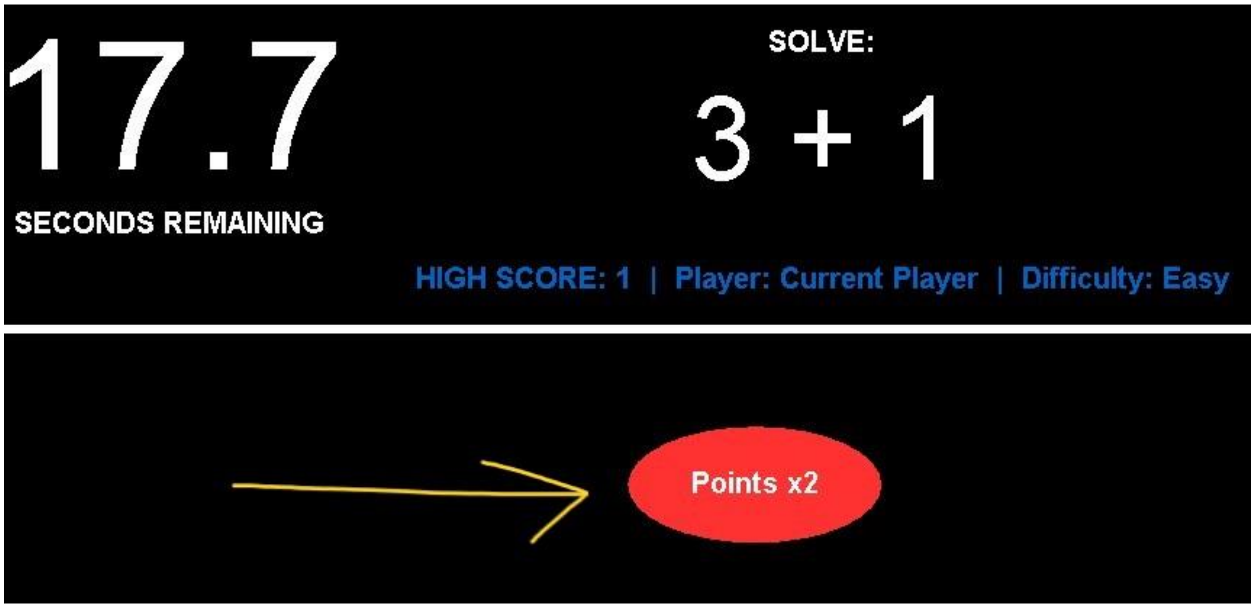
- **`x`** and **`y`** represent position.
- **`correct`** represents whether or not the answer choice is correct for the current equation.
- **`answer`** represents the answer (a double) stored in the current instance.
- **`answerAsString`** is **`answer`** converted to a String for easier use by Graphics.drawString(). This conversion occurs within the constructor, which also removes the decimal point information from the String if the number ends in ".0".
- **`dx`** and **`dy`** represent speed, and are randomly chosen between -1 and 1 upon instantiation.
- **`hitBox`** is a Rectangle object with x, y, width, and height based on the answer choice's x and y, and on the size of **`answerAsString`** on the screen. It is calculated with **`calculateHitbox()`**.

<u>The following methods are implemented in AnswerChoice:</u>

- **`public void calculateHitbox()`**: assigns an appropriate value to the **`hitBox`** instance variable based on **`x, y,`** and **`answerAsString`**.
- **`public boolean contains(Point p)`**: determines whether or not a supplied Point is within the hitbox.

- **`public void draw(Graphics g, Color c)`**: moves the drawing code into the object, cleaning up paintComponent. This method takes a Color object as a parameter for use with powerups.
- **`public void moveAnswer()`**: shifts the x and y position by dx and dy, respectively, and recalculates the hitbox by calling **`calculateHitbox()`**.
- **`public void checkOutOfBounds(GameWindow panel)`**: takes a GameWindow object (normally passed through as GameWindow.this) as a parameter in order to gauge the size of the GameWindow. The method then makes sure the current object is not out of bounds, adjusting it if so.

4. The Powerup Class

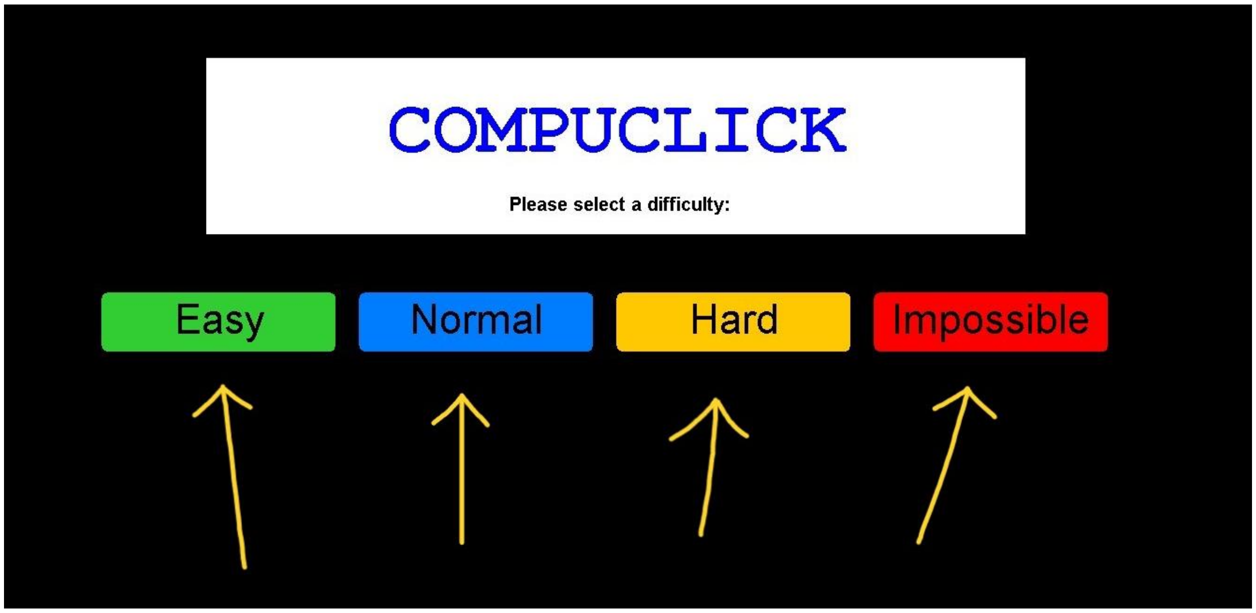


The **`Powerup`** class stores information about powerups that appear on screen. This includes position, color, and powerup type (0 represents slow down time, 1 represents double points, and 2 represents decreased equation difficulty).

`Powerup` instances are similar to **`AnswerChoice`** instances in that they both store x and y positions, implement a **`contains(Point p)`** method, implement a **`checkOutOfBounds(GameWindow panel)`** method, and implement a **`draw(Graphics g)`** method to simplify the code in **`GameWindow.paintComponent(Graphics g)`**.

Powerups do not use hitboxes, however, as they are represented as ellipses. As a result, the code for their **`contains()`** methods are quite different.

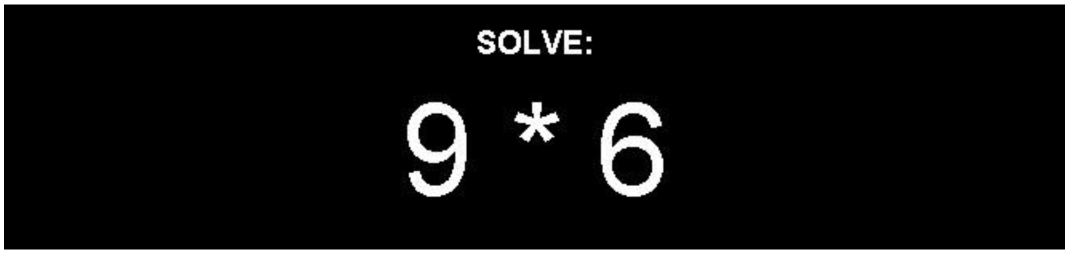
5. The ModeButton Class



The **`ModeButton`** class stores information about the buttons that appear on the title screen that allow players to select a game difficulty. This includes the button's label (either "Easy," "Normal," "Hard," or "Impossible), the button's difficulty as an integer (0 to 3), and the button's background color. All these values are assigned within the **`ModeButton`** constructor which takes **`x`**, **`y`**, and difficulty as an integer.

`ModeButton` instances are similar to **`AnswerChoice`** instances in that they store x and y positions, have a hitbox according to the text they display, implement a **`contains(Point p)`** method, and implement a **`draw(Graphics g)`**.

6. The Equation Class



The **`Equation`** class stores information about each round's equation that sits at the top of the screen. Each instance is a randomly generated equation of level 0, 1, or 2. Higher levels represent harder difficulties.

This class uses the following instance variables:

- **`equation`**: the String containing the randomly generated equation.
- **`answer`**: the answer to the equation, as computed by **`calculateAnswer()`**.
- **`answerAsString`**: the answer as a String, to be drawn with Graphics.drawString().
- **`hard`**: a boolean value that is set to true only when the equation is "hard." A "hard" equation is one containing trigonometric functions or calculus. These equations are hard coded into the game as an array, as they are too difficult to generate on the fly.

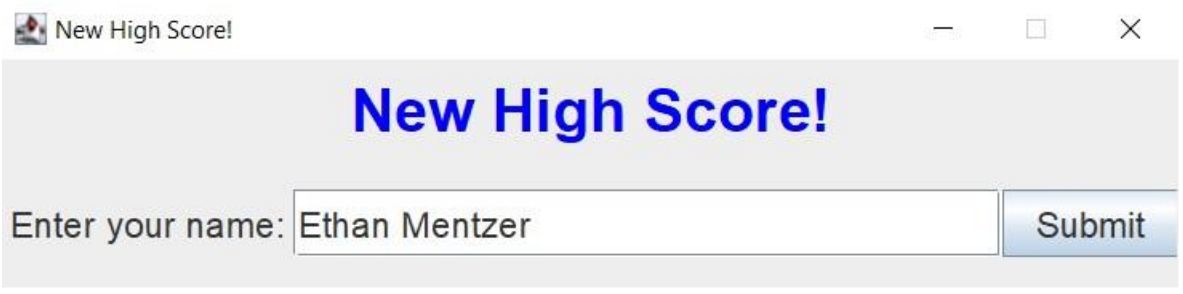
The **`Equation`** constructor takes one parameter, **`level`**, which determines its difficulty:

- Level 0 equations are two integers between 0 and 9 that are either added, subtracted, or multiplied.
- Level 1 equations are three integers between 0 and 9 that can be added, subtracted, or multiplied. Order of operations applies.
- Level 2 equations are two integers, the first from 0 to 9, and the second from 0 to 99, that can be added, subtracted, or multiplied. There is a 10% chance of receiving a "hard" problem containing trigonometric functions or calculus.

```
// Hard equations are hard-coded
String[] hardEqs = {"cos(π/3)", "sin(π)", "tan(π)", "f(x) = x^2, f'(2) = ?", "f(x) = ln(x), f'(1) = ?"};
double[] hardAns = {0.5, 0.0, 0.0, 4.0, 1.0};
```

The **`Equation`** class also implements the **`calculateAnswer()`** method. This method is used to calculate the answer to a randomly generated equation, which is non-trivial because Java cannot simply evaluate a mathematical expression String to produce a result. **`calculateAnswer()`** is only designed to handle the types of equations listed above, taking parameters for each number and operator in the equation. Method overloading is used to support both level 0 and 2 equations, while also supporting level 1 equations, which need to consider order of operations.

7. The HighScore Class


New High Score!

New High Score!

Enter your name: Ethan Mentzer

The **`HighScore`** class implements the ActionListener interface, and contains only static methods and instance variables, with the exception of actionPerformed(), which is non-static. The purpose of this class is as follows:

- To create a popup box in which players can enter their names if they score a high score.
- To read from a CSV file containing the high scores for each game difficulty level.

<u>These purposes are fulfilled by the following methods:</u>

- **`public static void newHighScore(int score, String difficulty)`**
    - Creates a popup GUI using a JFrame and nested layouts with JPanels.
    - The popup contains a text field and a submit button.
    - The text field and submit button both have the same action listener to process what the player has entered (actionPerformed() always sets the event's source to the text field).

- **`public void actionPerformed(ActionEvent e)`**
    - Opens a FileWriter that appends an entry to the "highScores.csv" file based on the user's input name.
    - Instantiates a PrintWriter with that FileWriter.
    - If they submit a blank name, the PrintWriter uses the name "Anonymous."
    - Uses PrintWriter to write "name,score,difficulty" to the CSV file.
    - Closes the PrintWriter and closes the popup GUI.

- **`public static Object[] getHighScore(int diff)`**
    - Searches the "highScores.csv" file for the most recent addition for the specified difficulty.
    - Returns an array of objects with a player name, their high score, and the difficulty they played at.
    - This method uses a FileReader and Scanners to parse the CSV data, using a comma as the delimiter.
    - Much of this method's code is wrapped in a try-with-resources block to catch potential errors with the file.

8. The WAVPlayer Class

The **`WAVPlayer`** class makes it easy to play a .wav audio file using the javax.sound packages. The **`WAVPlayer`** constructor takes two parameters: **`String fileName`** and **`boolean loop`**, assigning each value to instance variables of the same names.

The **`WAVPlayer`** class has two methods:

- **`public void playSound()`**
  - Locates the .wav file using Java's getClass() and getResources() methods based on **`fileName`**, assigning the resource to a URL from java.net.URL.
  - It then opens an AudioInputStream using that URL, and uses AudioSystem.getClip() to create a playable Clip object from the AudioInputStream.
  - The method sets the clip to loop continuously if the **`loop`** instance variable is true.
  - Finally, the method calls Clip.play() to play the audio.

- **`public void stopSound()`**
  - Simply runs Clip.stop() on the currently instantiated clip object.

NOTE: *ChatGPT helped me understand how to properly put together the code in* ***`WAVPlayer`***.