11/8/20: cutting out colorblind mode, pausing/changing volume in-game

|  |  |  |
| --- | --- | --- |
| **Class** | **Attributes** | **Methods** |
| **Score**   * Keep track of challenges (ex. 2/5 lines cleared, etc.) | * int score * int highScore | * **Score()**   + Initializes score to 0   + Initializes the highScore by reading from a file named highScore.txt * **void increaseScore(amount)**   + Adds amount to the score * **void saveHighScore()**   + If the score is greater than the current score, write the current score to a file as the new high school * setters/getters |
| **Block**   * NOTE: the values in the shape object in this class will likely change as the board processes matching rows and colors. * NOTE: first point of the shape is the first true in the shape array.   + Ex. the ‘s’ shape’s “first point” is at the top middle * NOTE: | * BlockColor color * Shape shape   + Should be one of the 2D array representations in one of the constant shape 3D arrays   + Example: o[1] * int points[8];   + Array of the coordinates of each square in the block   + Similar to how the coordinates in Polygon class work—points[0] and points[1] are x and y of the first coordinate, respectively * int orientation   + 0 (normal)   + 1 (rotate right once)   + 2 (rotate right twice)   + 3 (rotate right three times; aka rotate left from normal) | * **Block()**   + Initializes the color to 0   + Initializes shape to any shape (doesn’t matter, you choose)   + Initializes the first “point” in points[] to be as close to 0, 0 as possible while still having all the points be positive     - Example: T[1] would have the points (0, 1), (1, 0), (1, 1), (1, 2)     - Initialize the other points to something that makes spatial sense for the first point and the shape   + Initializes the orientation to 0 * **Block(BlockColor, Shape, int row, int col, int orientation)**   + x and y: first point of the shape; use to initialize the rest of the points[] * getters/setters for all variables; ones that are different than the standard getters:   + **Color getColor()**     - Returns the actual Color object that is currently represented   + **boolean[][] getShape()**     - Returns the shape[][] in the shape object   + **int getColorNum()**     - Returns the number that color is associated with (in the Block Color class)   + **Shape getShapeObj()**     - Returns the actual shape object * **void rotateRight()**   + Changes the orientation variable by increasing it (loop around if at end); update shape and points accordingly * **void rotateLeft()**   + Changes the orientation variable by decreasing it (loop around if at end); update shape and points accordingly * **void moveLeft()**   + adds 1 to y ~~if not at the leftmost column~~ * **void moveRight()**   + adds 1 to x ~~if not at the rightmost column~~ * **boolean isSameColor(otherBlock)**   + True if the other given block is the same color, false otherwise * **public void updatePoints(int newX, int newY)**   + Updates the points array by calculating the location of each square in the block based on the orientation, shape, and the new location (given by int x and int y variable)   + ~~Only for use within the class to use after rotating or moving~~   + Note: list points as if “reading” the shape from left to right, top to bottom.     - Example: points for ‘o’ shape at 0,0 should be:   0, 0,  0, 1,  1, 0,  1, 1   * **public boolean equals(Block)**   + Returns true if this block has the same attributes as the given block (addresses can be different, just the inherent values need to be the same). * **public int[] getFirstPoint()**   + Returns the row and column of the first point in terms of the shape array   + Returns {-1, -1} if no first point was found * **public int getFirstPointIndex()**   + Returns the index of the first valid point in the points array   + Returns -1 if no first point was found * **public boolean isValidShapePoint(int r, int c)**   + True if the given points are true in the block’s shape |
| **Board**   * Handles the play area only * Does not handle GUI, displaying block preview, etc. | * public staticfinal int ROWS = 20 * public static final int COLS = 10 * public static final int MAX\_BLOCKS = 50   + Calculate the total number of blocks that could possibly fit on a board with ROWS and COLS * private boolean board[ROWS][COLS] * private Block blocks[MAX\_BLOCKS]   + Contains all blocks that are currently on the board * private Block futureBlocks[Shape.NUM\_SHAPES]   + Contains the next seven blocks that will drop * private double normalBlockSpeed * private double dropBlockSpeed * private int squareSideLength   + Contains the length/width of a single square in pixels; use for drawing stuff on the board in the Game class * private int currentBlockIndex   + Contains the index of the current block in the futureBlocks array * private Block current   + Current block that the user is manipulating * private blocksLength;   + Used to keep track of the last element in the blocks array | * **Board()**   + Initialize board[][] with all false   + Initialize blocks[]   + Generate/initialize futureBlocks   + Initialize normalBlockSpeed to 0   + Initialize dropBlockSpeed to 0   + Initialize squareSideLength to 0   + Initialize currentBlockIndex to 0   + Initialize current to futureBlocks[0] * **Board(int normalSpeed, int dropSpeed, int sideLen)**   + Initialize board[][] with all false   + Initialize blocks[]   + Generate/initialize futureBlocks   + Initialize normalBlockSpeed to the given argument   + Initialize dropBlockSpeed to the given argument   + Initialize squareSideLength to the given argument   + Initialize currentBlockIndex to 0   + Initialize current to futureBlocks[0] * **void clearRow()**    + Checks each row and clears them if needed by setting the cleared portions in the shape to be false     - Also sets the points that are cleared in the block to -1   + May be helpful to use moveBlocksDown()   + Use this method when block hits the other stationary blocks * **void clearColors()**   + Checks for and clears color clusters   + May be helpful to use moveBlocksDown()   + Use this method when block hits the other stationary blocks * **void nextBlock()**   + Places the next block into the current variable   + Increments the currentBlockIndex variable   + Calls on generateNewFutureBlocks when needed * **boolean** **isValidLocation(int x, int y)**   + Checks to see if a point is on the board   + Use this to check if a user’s move would be valid or not * **void generateNewFutureBlocks()**   + Generates new blocks for the futureBlocks array.   + To make the game play more balanced:     - All shapes should be used at least once     - All colors should be used at least once     - The order and color-block assignment should be random * **void moveBlocksDown()**   + Checks to make sure that there are no blocks that are randomly floating   + If a block is not touching the bottom or resting somewhere on top of another block, it should “fall” down until it is resting on something * **void removeBlankBlocks()**   + Checks the blocks array and removes any blocks whose shapes are completely false * **void addBlock(Block b)**   + Adds b to the blocks array   + Adds b’s shape values to the board array * **void removeBlock(Block b)**   + Removes the first occurrence of b from the blocks array   + Removes b’s shape values from the board array * **void removeBlock(int)**   + Removes the block at the given index in the blocks array by swapping it with the last block * **void removeBlock(int x, int y)**   + Removes the block at the given coordinate * **Block[][] generateBlockLocations()**   + Returns an array in the same shape as the board with each Block’s address in the corresponding location * **Block[] getNeighbors(Block)**   + Returns an array of Blocks that are touching the given block * **boolean isAtBottom(Block)**   + Determines if the given block is resting on another block (true) or if it is floating (false) * getters/setters for all private variables:   + getters only for:     - board[][], blocks[], futureBlocks[]     - Getters will return the references of these arrays, so they can be modified like that without a setter |
| **Sound**   * Used to be the volume class. * For playing media files: <https://stackoverflow.com/questions/6045384/playing-mp3-and-wav-in-java> * Add other sound effects in the future? * Source for theme: <https://commons.wikimedia.org/wiki/File:Tetris_theme.ogg> | * double volume // percent volume; max is 1 * final double VOLUME\_CHANGE\_AMOUNT * static String music = “<music file name here>” * Media song; * MediaPlayer player; | * **Sound()**   + Initialize volume to 0.20   + Initialize song with default constructor for Media   + Initialize player with default constructor for MediaPlayer * **Sound(double)**   + Initialize volume to the given argument   + Initialize song with default constructor for Media   + Initialize player with default constructor for MediaPlayer * **void increaseVolume()**   + Increase volume by VOLUME\_CHANGE\_AMOUNT * **void decreaseVolume()**   + Decrease by VOLUME\_CHANGE\_AMOUNT * **void playSong()**   + Uses player and song objects to play the background music (fairly simple method—will be called on by Main) * getters/setters for volume, song, player |
| **~~BlockPreview~~**   * ~~Simple enough (and more convenient) to just implement in the Game class.~~ | * ~~Block preview[2]~~ | * **~~void addBlock(Block b)~~**   + ~~shifts the blocks up and adds the given block at the end~~ |
| **~~PowerUp~~** ~~(Block subclass)~~   * ~~Implement/decide implementation later, if we have time~~ | * ~~--~~ | * ~~--~~ |
| **BlockColor**   * Use the Color class in JavaFx | * public static final NUM\_COLORS = 7 * public static final Color COLORS[NUM\_COLORS] * ~~public static final Color COLORBLIND\_COLORS[NUM\_COLORS]~~ * ~~private static Color currentColors[]~~   + ~~Equal to COLORS or COLORBLIND\_COLORS~~ * private int color   + Contains the index of the current color | * **BlockColor()**   + Initialize color to 0 * **BlockColor(int)**   + Initialize color using the given int * **boolean isEqual(otherColor)** * **~~void toggleColoblindMode()~~**   + ~~Toggles the currentColors array (changes it to either COLOR or COLORBLIND\_COLORS based on what it is now)~~ * getters/setters for all the non-final variables   + NOTE: the getter for currentColors should be static. |
| **GameLauncher/Main** (initializes settings, calls menu, saves configurations)   * Programs start; displays game name, high score, play game, options, exit * If user selections options, transition to options page * Options   + Sound   + ~~Colorblind mode~~   + Return to menu | * Any variables needed to implement the main behaviors of Main (listed in left-most column) * Buttons for options on main menu, options page, game over page | * Any helper functions that help with the stuff Main needs to get done (person who writes this class gets to decide) * Inner class(es) to handle MouseEvents   + Program start     - Click on “New Game” button     - Click on “Options” button     - Click on “Exit”   + Options     - Click on “+” button to increase sound by some increment     - Click on “–” button to decrease sound by some increment     - Click on “Return to Menu” button   + Game over     - Click on “Play Again” button     - Click on “Exit to Main Menu” button |
| **Game** (runs the main game)   * Moves blocks * Gets user input * Updates with every cycle/frame * After the current block stops moving, checks for rows/colors that can be cleared * ~~Handles options when the game is paused~~ * Goes to game over screen when the game is over * Game over   + Current score   + High score   + Options to play again, exit to main menu * Updates the level (based on score) to increase difficulty. Not sure if we should display this or just use it internally. Might be good to not display it until this has more significance. * Handles GUI   + Displays instructions, score, preview of blocks * Handles user input and checks to see if actions can be done on the board | * final LEFT = KeyEvent.VK\_LEFT * final RIGHT = KeyEvent.VK\_RIGHT * final DROP = KeyEvent.VK\_DOWN * final ROTATE\_RIGHT = KeyEvent.VK\_Z * final ROTATE\_LEFT = KeyEvent.VK\_X * ~~final PAUSE = KeyEvent.VK.ESCAPE~~ * ~~int level // 1-3~~ * double speedMultiplier // depends on level * ~~double scoreMultiplier~~ * ~~LocalTime timePlayed~~ * challenges * boolean exitGame * Board board | * **Game()**   + ~~Initializes level to 0~~   + ~~Initialize timePlayed with the LocalTime constructor~~   + Initialize extiGame to false   + Initializes speed ~~and score~~ multipliers to 1   + Initializes board with the non-default constructor     - Pick variables that make sense—may need to play around with it a bit * Any helper functions needed (person who writes this class gets to decide)   + **public void playGame(Stage, Scene former)**     - Starts up the actual game; handles game functions and GUI     - Accepts the stage and the former scene   + **public int getScore()**     - Returns the int value of the game’s score object   + **private void setBoardPane(GridPane)**     - sets the given GridPane object to match the Game’s board object   + **private void setPreview(GridPane, Block)**     - Sets the given preview GridPane to the given Block   + **public boolean isAtBottom(Block)**     - Returns true if the block is resting at the bottom of the board or on another block, preventing it from falling   + **public boolean isValidMove(Block, KeyCode move)**     - Returns true if the block after performing the specified move is in the valid play area and not overlapping any blocks that are currently on the board       * move variable can be one of four constants: LEFT, RIGHT, ROTATE\_RIGHT, ROTATE\_LEFT   + **public boolean isValidBlock(Block)**     - Returns true if the block is on valid positions on the board   + **public void setBoardArray(Block, boolean)**     - Changes the board array to match the specified boolean for the points specified by the given block   + **public void moveDownOne(Block)**     - Moves the given block down by one index if it is not already at the bottom   + **private Rectangle newRectangle(int color)**     - Returns a new rectangle since javafx doesn’t allow duplicate things on one pane (and this program needs many of the same block) * Getters and setters for: ~~level~~, speedMultiplier, ~~scoreMultiplier~~, ~~timePlayed~~, exitGame, board * **~~void updatePreview()~~**   + ~~Uses the futureBlocks variable in board to update/display the preview of the next two blocks~~ * Inner class(es) to handle KeyEvents:   + Note: most of the KeyEvent objects that represent these keys are listed in the column to the left   + Move block right (right arrow key)   + Move block left (left arrow key)   + Drop block (down arrow key)   + Rotate block right (‘X’ key) (can change if group wants a different key)   + Rotate block left (‘Z’ key) (can change if group wants a different key)   + ~~Escape key to pull up menu~~ * ~~Inner class(es) for MouseEvents:~~   + ~~When paused:~~     - ~~Click on “+” to increase sound by some increment~~     - ~~Click on “–” to decrease the sound by some increment~~     - ~~Click on “Exit to Menu” to return to the main menu~~ |
| **Shape**   * Hardcodes each shape in an array along with each of their rotations * NOTE: first point of the shape is the first true in the shape array.   + Ex. the ‘s’ shape’s “first point” is at the top middle | * public final static boolean O[][][] = … * public final static boolean I[][][] = … * public final static boolean S[][][] = … * public final static boolean Z[][][] = … * public final static boolean L[][][] = … * public final static boolean J[][][] = … * public final static boolean T[][][] = … * public final static NUM\_SHAPES = 7;   + Total number of shapes that exist * private boolean shape[][]   + Shape for this object; should be one of the 2D arrays from the other static 3D arrays in this class * See bottom of the document for all of these attributes * private char letter;   + The letter name of the shape | * **Shape()**   + Initialize shape[][]to a COPY OF of the final static shape/orientation; doesn’t matter * **Shape(char, int)**   + Initializes shape[][] to a COPY OF the shape/orientation that corresponds to the given character and the given orientation (specified by the int) * **int getHeight()**   + Returns the height of shape/number of rows * **int getWidth()**   + Returns the width of the shape/number of columns * **void setShape(char, int)**   + Accepts a character corresponding to the block shapes (same as the final static variables in this class)   + Sets shape[][] to the corresponding final static 2D array in this class * **void setShape(int)**   + Keeps the ssame type of shape, but sets the shape[][] to the orientation specified by the given int * **boolean[][] getShape()**   + Returns shape[][] * Ones that were added during testing   + **private static int[] geDimensions(char shape, int orientation)**     - Returns an array with two elements; first element is the height and second element is the width.     - Used with the constructor for first determining the height/width |
| ~~Square~~   * ~~One square in a Block~~ * NOTE: decided not to do this. I think all the other information fits in other classes pretty well. |  |  |

NOTE: decided to make these arrays only large enough to fit each type of block (instead of 4x4) since it allows for easy access to the width and height of each block.

NOTE 2: sorry guys, I meant for the 1s and 0s below to be booleans (instead of ints) which should make it easier to code some things

NOTE 3: redefining rotations/shapes based on this: [www.gamedev.stackexchange.com/questions/17974/how-to-rotate-blocks-in-tetris](http://www.gamedev.stackexchange.com/questions/17974/how-to-rotate-blocks-in-tetris)

|  |  |  |  |
| --- | --- | --- | --- |
| final static boolean o[][][] = {  {{1, 1},  {1, 1}},  {{1, 1},  {1, 1}},  {{1, 1},  {1, 1}},  {{1, 1},  {1, 1}}  }; | final static boolean i[][][] = {  {{0, 0, 0, 0},  {1, 1, 1, 1},  {0, 0, 0, 0},  {0, 0, 0 ,0}},  {{0, 0, 1, 0},  {0, 0, 1, 0},  {0, 0, 1, 0},  {0, 0, 1, 0}},  {{0, 0, 0, 0}  {0, 0, 0, 0},  {1, 1, 1, 1},  {0, 0, 0, 0}},  {{0, 1, 0, 0},  {0, 1, 0, 0},  {0, 1, 0, 0},  {0, 1, 0, 0}}  }; | final static boolean s[][][] = {  {{0, 1, 1},  {1, 1, 0},  {0, 0, 0}},  {{0, 1, 0},  {0, 1, 1},  {0, 0, 1}},  {{0, 0, 0},  {0, 1, 1},  {1, 1, 0}},  {{1, 0, 0},  {1, 1, 0},  {0, 1,0}}  }; | final static boolean z[][][] = {  {{1, 1, 0},  {0, 1, 1}  {0, 0, 0}},  {{0, 0, 1},  {0, 1, 1},  {0, 1, 0}},  {{0, 0, 0}  {1, 1, 0},  {0, 1, 1}},  {{0, 1, 0},  {1, 1, 0},  {1, 0, 0}}  }; |
| final static boolean l[][][] = {  {{0, 0, 1},  {1, 1, 1},  {0, 0, 0}}  {{0, 1, 0},  {0, 1, 0},  {0, 1, 1}},  {{0, 0, 0},  {1, 1, 1},  {1, 0, 0}},  {{1, 1, 0},  {0, 1, 0},  {0, 1, 0}},  }; | final static boolean j[][][] = {  {{1, 0, 0},  {1, 1, 1}  {0, 0, 0}},  {{0, 1, 1},  {0, 1, 0},  {0, 1, 0}},  {{0, 0, 0},  {1, 1, 1},  {0, 0, 1}}  {{0, 1, 0},  {0, 1, 0},  {1, 1, 0}},  }; | final static boolean t[][][] = {  {{0, 1, 0},  {1, 1, 1}  {0, 0, 0}},  {{0, 1, 0},  {0, 1, 1},  {0, 1, 0}},  {{0, 0, 0},  {1, 1, 1},  {0, 1, 0}},  {{0, 1, 0},  {1, 1, 0},  {0, 1, 0}}  }; |  |