

Making 2048's GUI

IN JAVA SWING

Joe Pelz

COMM 2216 | 2014-12-05

Contents

Introduction	1
Making the Window	2
Creating the Frame	2
Adding a Panel	2
Adding Components	3
Laying it Out.....	4
Putting Game Panel above the Buttons.....	4
Arranging the Direction Buttons.....	4
Arranging the Game Panel.....	5
Creating A Custom Component	6
Custom Component for the Number Square.....	6
Drawing the Square	6
Centering the Text	7
Creating A Custom JPanel	8
Custom Panel to Hold the Game Grid.....	8
Adding the GamePanel to GameFrame	9
Interactivity – Action Listeners	10
Setting up for Interactivity.....	10
Making the Buttons Work.....	11
Making Keyboard Shortcuts.....	11
Pop-ups – Talking to the user	12
Adding a Game-Over popup	12
Adding a New Game popup	13
Glossary.....	14
Trouble-shooting Guide	15
Appendix A – Game Logic	15
GameCode class.....	15

Introduction

This instruction manual will guide you through several tasks in creating a GUI, with the end result of a playable 2048 clone.

This tutorial is designed for someone new to programming or new to using java but is capable of creating a class. The programmer should know what getters and setters are, be able to figure out how to create a new java.awt.**Dimension** object, and be able to resolve missing types by importing the appropriate library.

Throughout this manual, classes such as **JFrame** and **KeyEvent** are notated by bold-face font. Variables and methods names such as *panelMain* and *setPreferredSize* are written in italics.

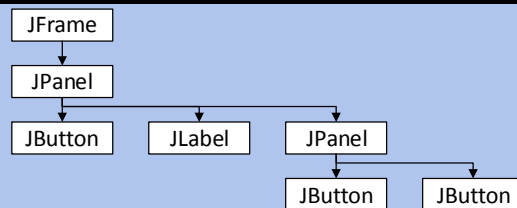
Making the Window

This section will cover how to get started in Java Swing. You will create a basic window, add a colored panel to it, and put components onto the panel.

NOTES

GUIs in swing

are all based around a **JFrame**.
JPanels are added to the **JFrame**,
and **JComponents** are added to Panels.



Creating the Frame

1. Create a new class called **GameFrame** that extends **JFrame**
2. Create a constructor that calls *super* with the title of your game as the parameter.
3. Create a public static void main method
4. Inside the main function,
 - a. Create a new **GameFrame** object called *frame*.
 - b. Set the default close operation (*setDefaultCloseOperation*) of *game* to **JFrame.EXIT_ON_CLOSE**. This ensures the program shuts down when you close the window.
 - c. Call *game*'s *pack* method to build it.
 - d. Set *game*'s visibility (*setVisible*) to true. This will "start" the program.



Adding a Panel

1. Define a *JPanel* instance variable named *panelMain*.
2. Initialize *panelMain* as a new **JPanel** in **GameFrame**'s constructor.
3. Give *panelMain* a background color (*setBackground*).
4. Add *panelMain* to **GameFrame**.



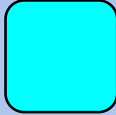
Colors can be created in many different ways.

There are several built-in presets accessed through the Color class, as `Color.CYAN` and `Color.ORANGE`

Colors can be created using hex codes like in html: `new Color(0xAACCFF)`

Colors can also be created with RGB values using floats between 0 and 1: `new Color(0.7f, 0.2f, 0.5f)`

Colors can be created using hue, saturation and brightness: `Color.getHSBColor(0.15f, 0.85f, 1.0f)`



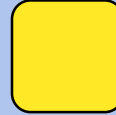
`Color.CYAN`



`new Color(0xAACCFF)`



`new Color(0.7f, 0.2f, 0.5f)`



`Color.getHSBColor(0.15f, 0.85f, 1.0f)`

Adding Components

1. Define another **JPanel** instance variable named *panelGame*.
2. Define four **JButton** instance objects named *btnLeft*, *btnRight*, *btnUp*, *btnDown*.
3. Instantiate *panelGame* in **GameFrame**'s constructor and add it to *panelMain*.
4. Give *panelGame* a preferred size (*setPreferredSize*) of 400 x 400. Use a new anonymous Dimension object.
5. Instantiate the buttons in **GameFrame**'s constructor with meaningful text such as "Slide Left".
6. Add the buttons to *panelMain*.
7. Now we're getting somewhere!



Laying it Out

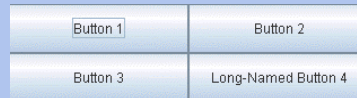
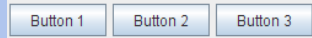
This section will cover using layouts to choose where components will be placed. You will place the game panel above the buttons, layout the buttons in a diamond shape, and then layout the game panel in a simple grid.

NOTES

There are many different layouts available.



BoxLayout (vertical and horizontal),



GridLayout(2, 2),



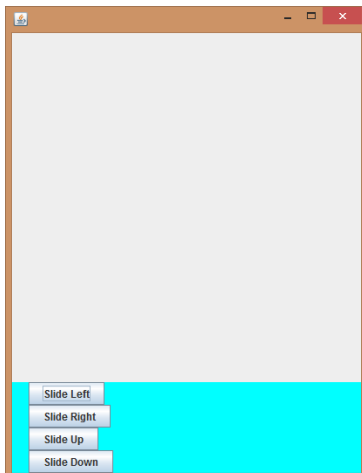
GridBagLayout

Many more layouts are available and Oracle has provided a reference here:

<http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html>

Putting Game Panel above the Buttons

1. Near *panelMain*, create a new **BoxLayout** using *panelMain* and **BoxLayout.Y_AXIS** as parameters
2. Apply the new layout to *panelMain* (using *setLayout*).



Arranging the Direction Buttons

1. Define and Initialize another **JPanel** instance variable named *panelButtons*.
2. Give *panelButtons* a new color (*setBackground*).
3. Set *panelButtons* layout (*setLayout*) to a new **GridBagLayout**.
4. Create a new local **GridBagConstraints** named *constraints*.
5. Remove the lines of code that add the buttons to *panelMain*.

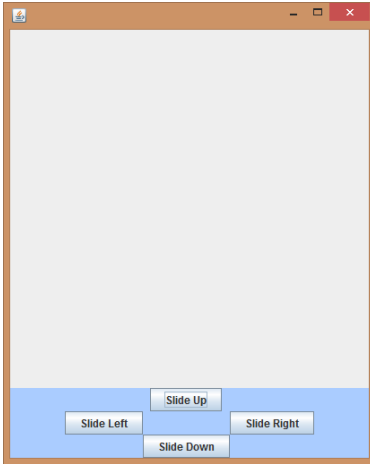
continued...

- For each button, set appropriate using *constraints.gridx* and *constraints.gridy*, and add it to *panelButtons* with *constraints* as a second parameter. Try to make it look like the image below.

```
GridBagConstraints constraints = new GridBagConstraints();
btnLeft = new JButton("Slide Left");
constraints.gridx = 0; constraints.gridy = 1;
panelButtons.add(btnLeft, constraints);

btnRight = new JButton("Slide Right");
constraints.gridx = 2; constraints.gridy = 1;
panelButtons.add(btnRight, constraints);
```

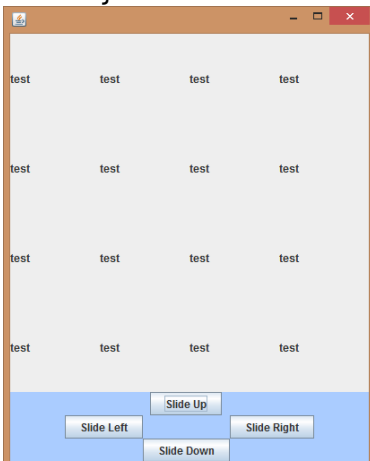
- Add *panelButtons* to *panelMain*.



Arranging the Game Panel

- Set *panelGame*'s layout to a new **GridLayout** with 4 columns and 4 rows.
- Use a for loop to add 16 anonymous JLabels to the *panelGame* for testing as follows:

```
panelGame = new JPanel();
panelGame.setLayout(new GridLayout(4, 4));
for (int i = 0; i < 16; i++) {
    panelGame.add(new JLabel("test"));
}
```



Creating A Custom Component

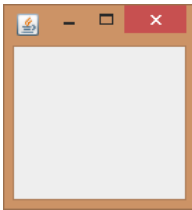
This section covers creating a custom component that is drawn in a custom way.

Custom Component for the Number Square

1. Create a **NumSquare** class that extends **JComponent**.
2. Define several static final variables:

Type	Name	Value
int	SCALE	100
int	BORDER	SCALE / 20
int	FONT_SIZE	(int)(SCALE * 0.4)
Font	FONT	new Font("Consolas", Font.PLAIN, FONT_SIZE)

3. Define a private integer instance variable named *value*.
4. Add a getter and a setter for *value*.
5. Create a constructor that takes one integer parameter.
6. In the constructor,
 - i. set *this.value* = *value*,
 - ii. set the font (*setFont*) to *FONT*,
 - iii. and set the preferred size (*setPreferredSize*) to *SCALE* x *SCALE*.



NOTES

Previewing your custom component

can be done by giving your class a main function to display it:

```
public static void main(String args[]) {  
    JFrame frame = new JFrame();  
    JPanel panel = new JPanel();  
    panel.add(new NumSquare(16));  
    frame.add(panel);  
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    frame.pack();  
    frame.setVisible(true);  
}
```

Drawing the Square

1. Override the *public void paintComponent(Graphics g)* method.
2. Use *getWidth*, *getHeight*, and *g.setColor* and *g.fillRect* to draw a white rectangle over the whole square
3. Change the color and use *fillRoundRect* to draw rounded shape inset by the value of *BORDER*. Use *SCALE / 3* as the radius.

4. Modify the color code for the *fillRoundRect* to use the following formula:

```
Color color;
if (value == 0) {
    color = Color.CYAN;
} else {
    int len = Integer.numberOfTrailingZeros(value);
    color = Color.getHSBColor(len / 11.0f, 0.8f, 0.5f);
}
g.setColor(color);
```

5. Set the color to light grey and use *drawString* to draw the **NumSquare**'s value in the center. Use *width / 2* and *height / 2* to find the center.
6. Fix the aliasing (the jagged, stepped edges) by adding the following command to the start of your *paintComponent* method:

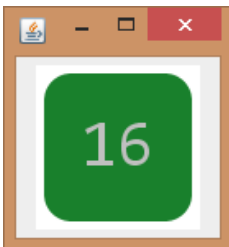
```
((Graphics2D)g).setRenderingHint(
    RenderingHints.KEY_ANTIALIASING,
    RenderingHints.VALUE_ANTIALIAS_ON);
```



Centering the Text

1. In *paintComponent*, before you call *drawString*, create a local **FontMetrics** object called *metrics*. Initialize it by assigning it the value of *getFontMetrics(FONT)*.
2. Use the *getAscent* method of *metrics* to offset the y position of your string by one third its height in *drawString*.
3. Use the *stringWidth* method of *metrics* to offset the x position of your string by half its width in *drawString*.

```
String txt = Integer.toString(value);
g.drawString(txt,
    (getWidth() - metrics.stringWidth(txt)) / 2,
    getHeight() / 2 + metrics.getAscent() / 3);
```



Creating A Custom JPanel

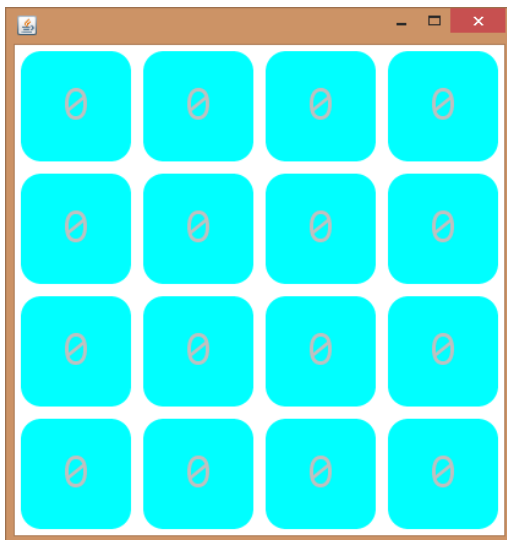
This section covers creating a custom panel to store a grid of numbers. It is the main game grid the user interacts with.

Custom Panel to Hold the Game Grid

1. Create a new class named **GamePanel** that extends **JPanel**.
2. Give **GamePanel** two integer variables named *COLUMNS* and *ROWS*.
3. Give **GamePanel** a 2D array of **NumSquares** named *numbers*.
4. Create a method named *init* that takes two parameters: *xSize* and *ySize*. *init* should initialize the game grid as follows:

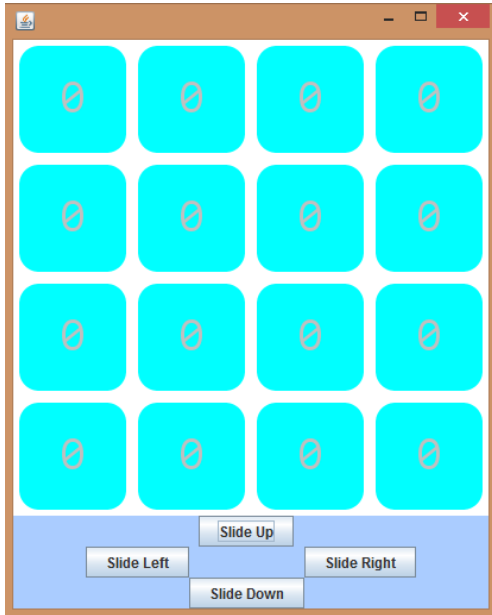
```
removeAll();
COLUMNS = xSize;
ROWS = ySize;
setLayout(new GridLayout(ROWS, COLUMNS));
numbers = new NumSquare[COLUMNS][ROWS];
for (int row = 0; row < ROWS; row++) {
    for(int col = 0; col < COLUMNS; col++) {
        numbers[col][row] = new NumSquare(0);
        add(numbers[col][row]);
    }
}
```

5. Create a constructor that takes two parameters: *xSize* and *ySize*. The constructor should call *init* with *xSize* and *ySize* as parameters
6. Create a getter and a setter for *numbers*, that takes two integers, column and row. Name them *getValue* and *setValue*. They should set or return the value of the **NumSquare** at the given coordinates in the *numbers* array.



Adding the GamePanel to GameFrame

1. In **GameFrame**, change the type of *panelGame* from **JPanel** to **GamePanel**. Use 4 for the rows and columns.
2. Remove the for loop that added **JLabels** to **panelGame**.
3. Remove the preferred size for **panelGame**.
4. Remove the layout assignment from **panelGame**.



NOTES

Extending Everything

Any class in Java's swing framework can be extended based on the needs of the user.

- We extended **JComponent** because we needed nothing more than a component with some drawing code and a value.
- Then we extended **JPanel** because we needed a component that would hold a grid of numbers for us.

Any component can be extended and any default behaviour can be overridden. You can extend **JButton** and change the drawing code so that it can still properly handle clicks. Or you could change the *setText* method of a text field so that it will only accept validated strings.

Interactivity – Action Listeners

This section covers interaction by using event listeners. You will add action listeners to the buttons, so that the buttons actually do stuff when clicked, and you will add key listeners so that you can press the arrow keys instead of clicking the buttons.

Important:

This section requires the `GameCode` class found in [Appendix A](#). The logic of 2048 is not the focus of this manual so the code to perform the actions required by the game is provided in standalone format.

Setting up for Interactivity

1. Create a new method called `updateNumSquares` in **GameFrame** that returns void and takes no arguments.
2. In `updateNumSquares`, use two nested for loops to iterate over all the rows and columns of the game grid. Update the values of `numbers` with the values from `game`. Use `panelGame.setValue` and `game.getCellValue`.
3. Call `panelGame.repaint` at the end of `updateNumSquares` to update what is displayed on the screen.
4. Add a **GameCode** instance variable named `game` to **GameFrame**.
5. In **GameFrame**'s constructor, before `panelGame` is initialized, initialize `game` to a new **GameCode** of size (4, 4).
6. Change `panelGame`'s initialization to use `game.COLUMNS` and `game.ROWS` as parameters.
7. Add two calls to `game.addNew2`, and one to `updateNumSquares` to the end of **GameFrame**'s constructor.



Making the Buttons Work

1. Add an anonymous **ActionListener** to *btnLeft*.
2. In the *ActionPerformed* method:
 - a. Call *game.slideLeft* to push all the tiles to one side.
 - b. Call *game.addNew2* to add another tile to the board.
 - c. Call *updateNumSquares* to update the values in *panelGame*.
3. Repeat for each of the directions, calling the appropriate slide method.
4. Call *panelGame.repaint()* to update what is displayed on the screen.



Making Keyboard Shortcuts

1. In **GameFrame**'s constructor, add an anonymous **KeyListener** using *addKeyListener*.
2. In the **KeyListener**'s *keyPressed* method, test if *e.getKeyCode() == KeyEvent.VK_UP* and if so, call *game.slideUp*, *game.addNew2*, and *updateNumSquares*.
3. Repeat with else-if statements for the other directions.
4. Set the focusable attribute (*setFocusable*) of all four buttons to false.

NOTES

Using modifier keys in shortcuts

You can test in your keypress handling code whether certain keys are already pressed. For example if you wanted to check for ctrl-n your if statement would resemble:

```
if (e.getKeyCode() == KeyEvent.VK_N && e.isShiftDown())
```

There are built in functions for ctrl, alt, shift, and meta being down. If you wanted to test if an arbitrary key, such as spacebar, were pressed, you would have to program the logic yourself.

Pop-ups – Talking to the user

This section implements dialog boxes, to let the user know their score, and to allow the user to choose the size of a new game before starting.

Adding a Game-Over popup

1. Create two new methods in **GameFrame** called *gameOver* and *newGame* that take no parameters and return void.
2. In *gameOver*, create a **String** array named *options* holding “New Game” and “Exit”
3. Create a message box as follows:

```
int result = JOptionPane.showOptionDialog(  
    this,  
    "Game over.\nYour score was " + game.getScore(),  
    "Game Over!",  
    JOptionPane.YES_NO_OPTION,  
    JOptionPane.INFORMATION_MESSAGE,  
    null,  
    options,  
    options[0]);
```

4. If *result* equals **JOptionPane.YES_OPTION**, call *newGame*.
5. Otherwise call **System.exit**.
6. At the end of the *updateNumSquares* method, add a check for game over: if *game.canPlay* is false, call *gameOver*.



Adding a New Game popup


1. In *newGame*, create **String** array named *options* holding "3x3", "4x4", and "5x5".
2. Create a message box as follows.

```
String choice = (String)JOptionPane.showInputDialog(  
    this,  
    "What size game field?",  
    "New Game",  
    JOptionPane.PLAIN_MESSAGE,  
    null,  
    options,  
    options[1]);
```

3. If *choice* is null, return early.
4. Otherwise, assign *game* to a new **GameCode** of size (3, 3), (4, 4), or (5, 5) based on the value of *choice*.
5. Outside of the if statement, Call *game.addNew2* twice.
6. Call *updateNumSquares*.
7. Call *pack*.



Glossary

Action Listener	A class with one method to be triggered when the component is activated, such as a button being clicked. See Event Listener for more information.
Aliasing, Anti-aliasing	Aliasing, also known as crunchy or jagged edges, is a problem that occurs when rendering pictures. Anti-aliasing uses semi-transparent pixels to fix it. See below:  The image shows two lowercase 'a' characters side-by-side. The first 'a' is labeled 'Alias' and has a jagged, pixelated appearance. The second 'a' is labeled 'Anti-aliased' and has smooth, rounded edges with some transparency at the boundaries.
Component	Any element of a GUI other than the frame. Panels, buttons, labels, checkboxes, textfields and menus are all components. A layout is not a component.
Event Listener	An event listener is a method that is triggered when a change happens to some other things the event listener is “watching.” All interaction in a java swing application is based on events. When a button is clicked, a mouse is scrolled, a key is pressed, the window is minimized or anything else, an event is fired. If an event listener is watching for that particular event, the listener will run it’s method.
Focusable	Whether the given component is allowed to have ‘focus.’ When a component has focus, it intercepts most keyboard events exclusively. The tab key cycles the focus through focusable components in a UI. Disabling the “focusable” attribute means that a component can no longer intercept keyboard events.
Getter (and setter)	Getters and setters are simple methods that provide controlled access to private variables that would otherwise be inaccessible. The getter facilitates reading and the setter facilitates writing.
GUI	Graphical User Interface. A GUI is the visible part of your program that the user sees.
Instance Variable	A variable or constant that is a member of a class, and not local to a method or temporary block.
Key Listener	A class with methods to triggered by keyboard events such as key presses. See Event Listener for more information.
Layout	A javax.swing class that provides control over the layout of a JPanel. Many varieties exist and a visual guide is available from Oracle’s website: http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html
Setter (and getter)	Getters and setters are simple methods that provide controlled access to private variables that would otherwise be inaccessible. The getter facilitates reading and the setter facilitates writing.

Trouble-shooting Guide

Trouble	Resolution
<code>panelGame</code> [or other variable] cannot be resolved	Ensure that the variable is an instance variable, and not a local variable where it is defined.
<code>GameCode</code> cannot be resolved to a type	<code>GameCode</code> is the logic for the game 2048. It is supplied in Appendix A and must be added to your project.
[class] cannot be resolved to a type	The class has not been imported. All classes used in this manual can be found in one of the following: <code>javax.swing.*</code> <code>java.awt.event.*</code> <code>java.awt.*</code>
My direction buttons aren't being placed correctly	Make sure you set <code>constraints.gridx</code> and <code>constraints.gridy</code> immediately before you add the button. When you add the button, include the constraints. <code>add(btnLeft, constraints);</code>
My numbers look different / My numbers are in the wrong font	You may not have the font "Consolas" installed, in which case the default font would be used. You can replace "Consolas" with the name of any installed font.
My keyboard shortcuts aren't working	Usually this is because a component has focus. Double check that each of your buttons has been made unfocusable via <code>setFocusable(false);</code>

Appendix A – Game Logic

GameCode class

```
import java.util.Random;

/**
 * This class provides the logic for 2048
 * with no interface.
 * Included is a printGame method that
 * will display the values in the internal array.
 *
 * @author Joe Pelz
 * @version 1.0
 */
public class GameCode {
    /** The number of columns in the game grid. */
    public final int COLUMNS;
    /** The number of rows in the game grid. */
    public final int ROWS;

    /** Grid holds the array of game numbers.
     * It is 2 elements larger than COLUMNS and ROWS in order to be
```



```

    * surrounded by a ring of extra zeroes.
    * <p>This ring of zeroes mostly just simplifies canPlay().</p> */
private int[][] grid;

/**
 * printGame() will print out the contents of the internal table
 * to stdout, representing the current game state.
 */
public void printGame() {
    for (int row = 1; row <= ROWS; row++) {
        for (int col = 1; col <= COLUMNS; col++) {
            System.out.printf("%d\t", grid[col][row]);
        }
        System.out.println();
    }
}

/**
 * Initialize a new game of the given dimensions.
 * Once set, the dimensions cannot be modified.
 *
 * @param columns The number of columns in the game grid.
 * @param rows The number of rows in the game grid.
 * @throws IllegalArgumentException Thrown if rows or columns are < 2
 */
public GameCode(final int columns, final int rows) throws IllegalArgumentException {
    if (columns < 2 || rows < 2)
        throw new IllegalArgumentException("Rows and columns must both be >= 2.");
    this.COLUMNS = columns;
    this.ROWS = rows;
    grid = new int[columns + 2][rows + 2];
    for (int col = 0; col < COLUMNS + 2; col++) {
        for (int row = 0; row < ROWS + 2; row++) {
            grid[col][row] = 0;
        }
    }
}

/**
 * Get the value of a particular cell in the 2D game grid array.
 *
 * @param col The column to query in the game grid.
 * @param row The row to query in the game grid.
 * @return The value of the given array cell.
 * @throws IndexOutOfBoundsException If the row or column is outside the game range.
 */
public int getCellValue(int col, int row) throws IndexOutOfBoundsException {
    if (col < 0 || col >= COLUMNS || row < 0 || row >= ROWS)
        throw new IndexOutOfBoundsException();
    return grid[col+1][row+1];
}

/**
 * Set the value of a particular cell in the 2D game grid array.
 *
 * @param col The column to target in the game grid.
 * @param row The row to target in the game grid.
 *
 * @throws IndexOutOfBoundsException If the row or column is outside the game range.
 */
public void setCellValue(int col, int row, int value) throws IndexOutOfBoundsException {
    if (col < 0 || col >= COLUMNS || row < 0 || row >= ROWS)
        throw new IndexOutOfBoundsException();
    grid[col+1][row+1] = value;
}

/**
 * Push all grid values to the left, filling in empty squares (0s).
 * If two equal numbers are pushed together,
 * they will combine to a larger number, double the size.
 */

```

```

public void slideLeft() {
    int destCol;
    for (int row = 1; row <= ROWS; row++) {
        destCol = 1;
        for (int column = 2; column <= COLUMNS; column++) {
            if (destCol == column || grid[column][row] == 0) {
                continue;
            } else if (grid[column][row] == grid[destCol][row]) {
                grid[destCol][row] = grid[destCol][row] * 2;
                grid[column][row] = 0;
                destCol++;
            } else {
                if (grid[destCol][row] != 0)
                    destCol++;
                if (destCol != column) {
                    grid[destCol][row] = grid[column][row];
                    grid[column][row] = 0;
                }
            }
        }
    }
}

/**
 * Push all grid values to the right, filling in empty squares (0s).
 * If two equal numbers are pushed together,
 * they will combine to a larger number, double the size.
 */
public void slideRight() {
    int destCol;
    for (int row = 1; row <= ROWS; row++) {
        destCol = COLUMNS;
        for (int column = COLUMNS - 1; column >= 1; column--) {
            if (destCol == column || grid[column][row] == 0) {
                continue;
            } else if (grid[column][row] == grid[destCol][row]) {
                grid[destCol][row] = grid[destCol][row] * 2;
                grid[column][row] = 0;
                destCol--;
            } else {
                if (grid[destCol][row] != 0)
                    destCol--;
                if (destCol != column) {
                    grid[destCol][row] = grid[column][row];
                    grid[column][row] = 0;
                }
            }
        }
    }
}

/**
 * Push all grid values to the top, filling in empty squares (0s).
 * If two equal numbers are pushed together,
 * they will combine to a larger number, double the size.
 */
public void slideUp() {
    int destRow;
    for (int column = 1; column <= COLUMNS; column++) {
        destRow = 1;
        for (int row = 2; row <= ROWS; row++) {
            if (destRow == row || grid[column][row] == 0) {
                continue;
            } else if (grid[column][row] == grid[column][destRow]) {
                grid[column][destRow] = grid[column][destRow] * 2;
                grid[column][row] = 0;
                destRow++;
            } else {
                if (grid[column][destRow] != 0)
                    destRow++;
                if (destRow != row) {
                    grid[column][destRow] = grid[column][row];
                    grid[column][row] = 0;
                }
            }
        }
    }
}

```

```

        grid[column][destRow] = grid[column][row];
        grid[column][row] = 0;
    }
}

}

}

/**
 * Push all grid values to the bottom, filling in empty squares (0s).
 * If two equal numbers are pushed together,
 * they will combine to a larger number, double the size.
 */
public void slideDown() {
    int destRow;
    for (int column = 1; column <= COLUMNS; column++) {
        destRow = ROWS;
        for (int row = ROWS - 1; row >= 1; row--) {
            if (destRow == row || grid[column][row] == 0) {
                continue;
            } else if (grid[column][row] == grid[column][destRow]) {
                grid[column][destRow] = grid[column][destRow] * 2;
                grid[column][row] = 0;
                destRow--;
            } else {
                if (grid[column][destRow] != 0)
                    destRow--;
                if (destRow != row) {
                    grid[column][destRow] = grid[column][row];
                    grid[column][row] = 0;
                }
            }
        }
    }
}

/**
 * Place a new 2 at a random empty place in the game grid.
 * An empty place is any cell with a value of 0
 *
 * @return
 */
public boolean addNew2() {
    int col;
    int row;
    Random random = new Random();

    if (isFull()) {
        return false;
    }

    do {
        col = random.nextInt(COLUMNS) + 1;
        row = random.nextInt(ROWS) + 1;
    } while (grid[col][row] != 0);

    grid[col][row] = 2;
    return true;
}

/**
 * Determine if there are any possible moves left.
 *
 * @return True if there are empty spaces or it is possible to combine two cells
 */
public boolean canPlay() {
    if (!isFull())
        return true;

    for (int col = 1; col <= COLUMNS; col++) {

```

```

        for (int row = 1; row <= ROWS; row++) {
            if (grid[row][col] == grid[row + 1][col]
                || grid[row][col] == grid[row][col + 1]
                || grid[row][col] == grid[row - 1][col]
                || grid[row][col] == grid[row][col - 1])
                return true;
        }
    }
    return false;
}

/**
 * Tally the values of all cells in the game grid.
 *
 * @return The sum of all cells in the game grid.
 */
public int getScore() {
    int score = 0;
    for (int col = 1; col <= COLUMNS; col++) {
        for (int row = 1; row <= ROWS; row++) {
            score += grid[col][row];
        }
    }
    return score;
}

/**
 * Check if the game grid is full (i.e. there are no more empty spaces or zeroes.)
 *
 * @return True if there is no more empty (0) cells in the game grid.
 */
public boolean isFull() {
    for (int column = 1; column <= COLUMNS; column++) {
        for (int row = 1; row <= ROWS; row++) {
            if (grid[column][row] == 0) {
                return false;
            }
        }
    }
    return true;
}
}

```