QUIZ 2

IF184401 Design & Analysis of Algorithms (H)

Hanafi Satriyo Utomo Setiawan (5025211195)

M. Armand Giovani (5025211054)

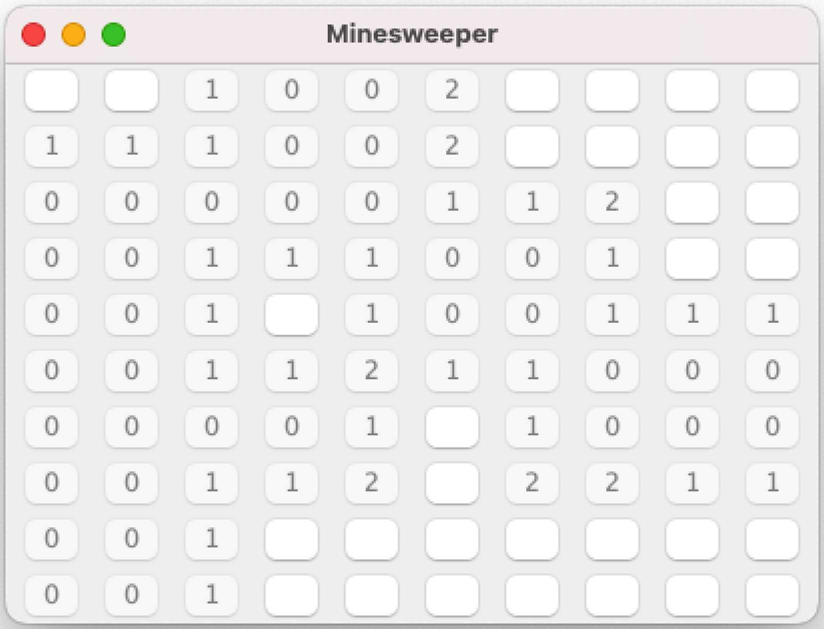
Christian Kevin Emor (5025211153)

DAA H

[All] Start From Here

Question : You are free to make any computer program for your group project (e.g., game, start-up, etc.). However, you have to implement any algorithms (e.g., DFS, BFS, DAG, Prim-Jarnik, Kruskal, etc.) that have been taught in our lectures. For instance, a game that determines the closest distance, a minesweeper game or a web application that examines the minimum distance for sending the goods from one point to another point.

Answer : In this assignment, we have been tasked with developing a minesweeper game, which involves creating a digital application that simulates the classic puzzle game where players uncover hidden mines on a grid-based playing field. Our objective is to design and implement the game mechanics, including generating a random minefield, placing mines strategically, and incorporating user interactions such as revealing cells, flagging potential mines, and calculating the number of adjacent mines. Throughout the development process, we will also ensure an intuitive user interface, smooth gameplay experience, and appropriate visual and audio elements to enhance engagement and enjoyment for players.



		1	0	0	2				
1	1	1	0	0	2				
0	0	0	0	0	1	1	2		
0	0	1	1	1	0	0	1		
0	0	1		1	0	0	1	1	1
0	0	1	1	2	1	1	0	0	0
0	0	0	0	1		1	0	0	0
0	0	1	1	2		2	2	1	1
0	0	1							
0	0	1							

Minesweeper is a classic puzzle game where the objective is to uncover all the empty cells on a grid-based playing field without detonating any hidden mines. The game typically starts with a blank grid, and the player must strategically click on cells to reveal their contents. Each revealed cell either contains a mine or a number indicating the number of adjacent cells that contain mines. By deducing the location of the mines based on the revealed numbers, the player can make calculated moves to avoid mines and gradually clear the entire grid. The challenge lies in making logical deductions and employing a process of elimination to navigate through the minefield successfully. Minesweeper requires a combination of skill, strategy, and critical thinking to win.

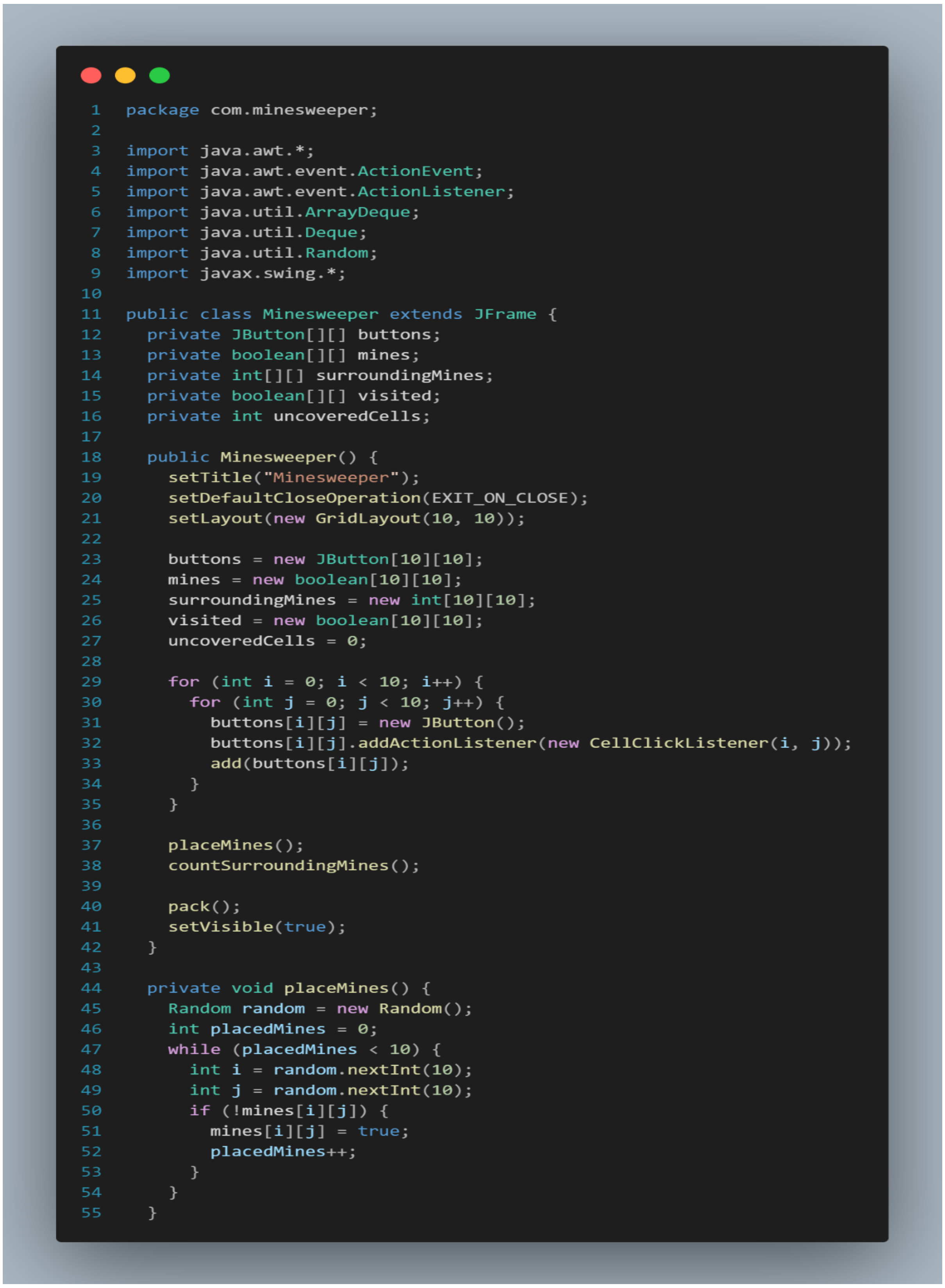
Uses a recursion algorithm (DFS - Depth-First Search) to reveal connected cells in the game Minesweeper. This algorithm is used in the uncoverSurroundingCellsDFS method which unravels cells and iteratively explores connected cells that have no mines around them. This algorithm utilizes the stack to track the cells to be checked next. In addition, this code also uses a random algorithm (Random) to place mines in the placeMines method.

The algorithm used in this code includes the following steps:

1. Initialization: It sets up the game window, creates the necessary arrays, and initializes variables.
2. Placing Mines: Randomly selects and places 10 mines on the grid.
3. Counting Surrounding Mines: Calculates the number of mines adjacent to each cell and stores the count in the surroundingMines array.
4. Cell Uncovering: Handles the logic when a cell is uncovered by the player. If the cell contains a mine, the player loses the game. Otherwise, the number of surrounding mines is displayed on the button, and if the cell has no surrounding mines, the algorithm recursively uncovers adjacent cells using Depth-First Search (DFS).
5. DFS for Uncovering Surrounding Cells: Uses a stack to perform DFS on neighboring cells and uncover them until cells with surrounding mines are encountered.
6. Win or Lose: Displays a message box indicating whether the player has won or lost the game and exits the program accordingly

[M. Armand Giovani / 5025211054] Start From Here

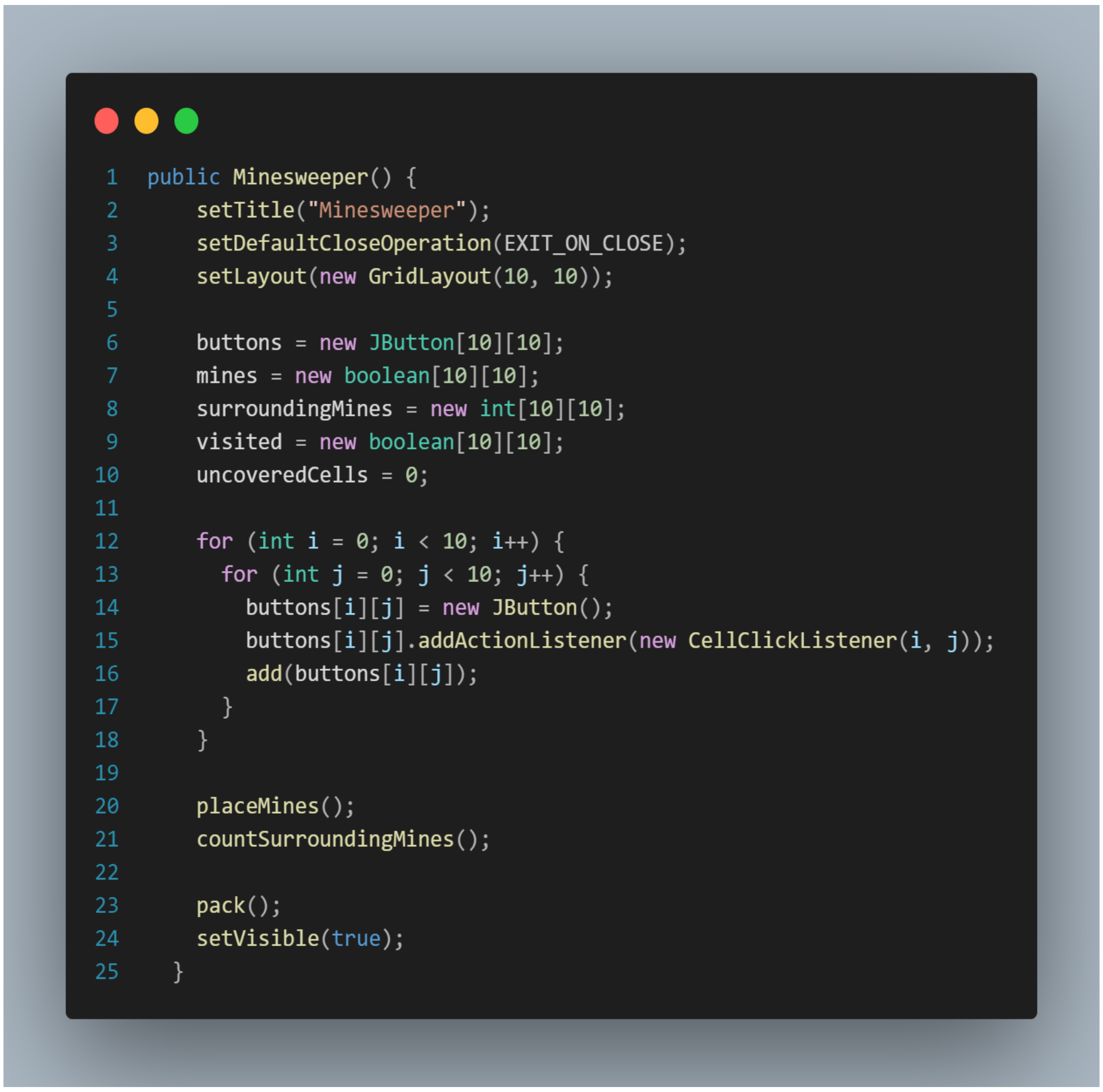
Class Minesweeper



```java
package com.minesweeper;

import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.Random;
import javax.swing.*;

public class Minesweeper extends JFrame {
  private JButton[][] buttons;
  private boolean[][] mines;
  private int[][] surroundingMines;
  private boolean[][] visited;
  private int uncoveredCells;

  public Minesweeper() {
    setTitle("Minesweeper");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setLayout(new GridLayout(10, 10));

    buttons = new JButton[10][10];
    mines = new boolean[10][10];
    surroundingMines = new int[10][10];
    visited = new boolean[10][10];
    uncoveredCells = 0;

    for (int i = 0; i < 10; i++) {
      for (int j = 0; j < 10; j++) {
        buttons[i][j] = new JButton();
        buttons[i][j].addActionListener(new CellClickListener(i, j));
        add(buttons[i][j]);
      }
    }

    placeMines();
    countSurroundingMines();

    pack();
    setVisible(true);
  }

  private void placeMines() {
    Random random = new Random();
    int placedMines = 0;
    while (placedMines < 10) {
      int i = random.nextInt(10);
      int j = random.nextInt(10);
      if (!mines[i][j]) {
        mines[i][j] = true;
        placedMines++;
      }
    }
  }
```

The provided code snippet contains the following functions:

1. **Minesweeper()** constructor is responsible for initializing the game and setting up the user interface. Here are the main tasks performed in the constructor:

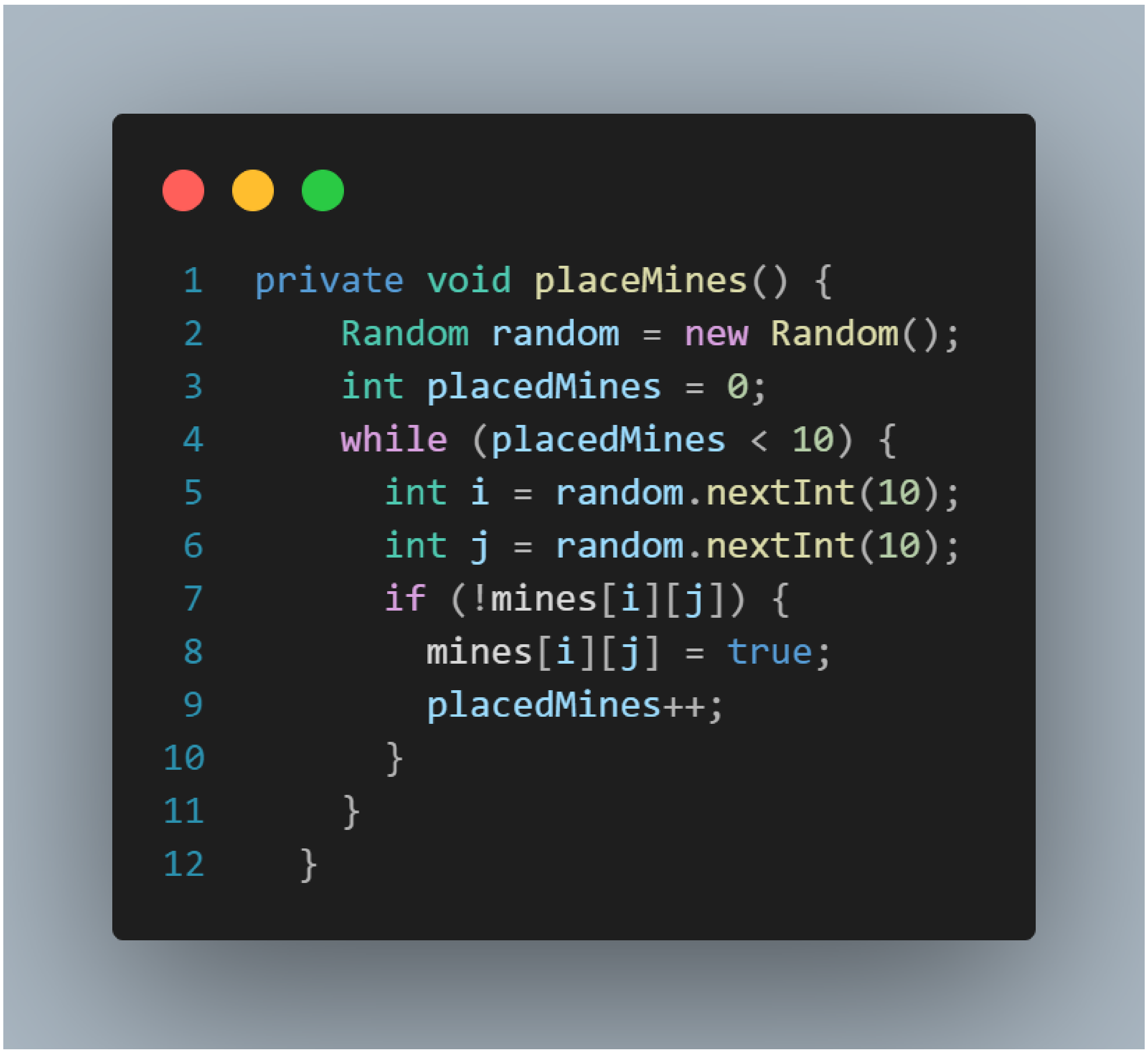


```java
public Minesweeper() {
    setTitle("Minesweeper");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setLayout(new GridLayout(10, 10));

    buttons = new JButton[10][10];
    mines = new boolean[10][10];
    surroundingMines = new int[10][10];
    visited = new boolean[10][10];
    uncoveredCells = 0;

    for (int i = 0; i < 10; i++) {
      for (int j = 0; j < 10; j++) {
        buttons[i][j] = new JButton();
        buttons[i][j].addActionListener(new CellClickListener(i, j));
        add(buttons[i][j]);
      }
    }

    placeMines();
    countSurroundingMines();

    pack();
    setVisible(true);
  }
```

- Set the title of the game window to "Minesweeper" using setTitle("Minesweeper").
- Set the default close operation for the game window to exit the application when the window is closed using setDefaultCloseOperation(EXIT_ON_CLOSE).
- Set the layout manager of the game window to a 10x10 grid layout using setLayout(new GridLayout(10, 10)).
- Create the necessary arrays (buttons, mines, surroundingMines, visited) to store information about the game board, mines, and other game-related data.

- Use nested for loops to create JButton objects for each cell on the game board, add an ActionListener to handle button clicks, and add the buttons to the game window.
- Call the placeMines() method to randomly place mines on the game board.
- Call the countSurroundingMines() method to calculate the number of surrounding mines for each cell.
- Adjust the size of the game window to fit its contents using pack().
- Set the game window to be visible to the user using setVisible(true).

2. **placeMines()** method is used to randomly place mines on the game board. Here is an explanation of the steps performed in this method:

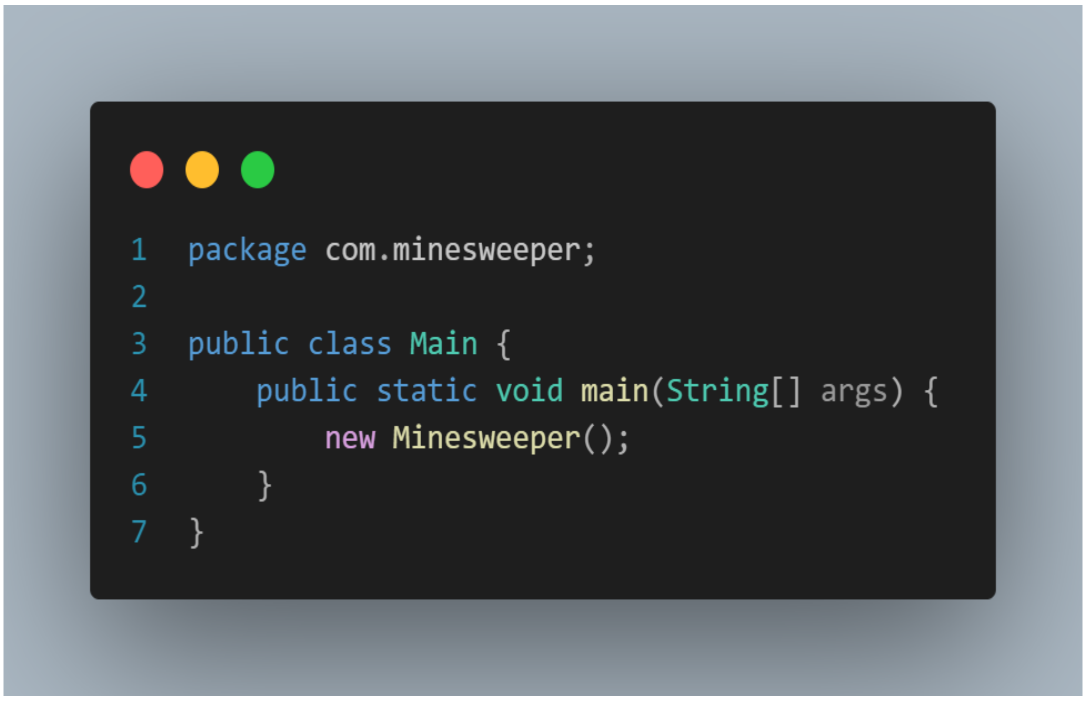


```java
private void placeMines() {
    Random random = new Random();
    int placedMines = 0;
    while (placedMines < 10) {
      int i = random.nextInt(10);
      int j = random.nextInt(10);
      if (!mines[i][j]) {
        mines[i][j] = true;
        placedMines++;
      }
    }
  }
```

- Create a Random object to generate random numbers.
- Declare the placedMines variable with an initial value of 0. This variable will be used to count the number of mines that have been placed.

- Enter a while loop as long as the number of placed mines (placedMines) is less than 10 (the desired number of mines to be placed).
- In each iteration of the loop, generate two random numbers (i and j) using random.nextInt(10). The number 10 is used because we have a game board with a size of 10x10.
- Check if the cell at coordinates (i, j) does not have a mine (mines[i][j] is false).
- If the cell does not have a mine, mark that cell as having a mine (mines[i][j] = true) and increment the count of placed mines (placedMines++).
- The loop continues until the number of placed mines reaches 10.

Class Main



```java
package com.minesweeper;

public class Main {
    public static void main(String[] args) {
        new Minesweeper();
    }
}
```

The provided code snippet contains the following functions:

1. Package name suggests that it is related to the Minesweeper game project.
   ```java
   package com.minesweeper;
   ```
2. public class Main {}

```java
public class Main {

  public static void main(String[] args) {

      new Minesweeper();
```

```
    }

}
```

- The **Main** class is declared as **public**, meaning it can be accessed from other classes.
- The **main** method is the entry point of the program and serves as the starting point for the execution.
- Inside the **main** method, a new instance of the **Minesweeper** class is created using the new keyword.
- This line **new Minesweeper()**; initializes and starts the Minesweeper game by invoking the constructor of the **Minesweeper** class.

The purpose of this **Main** class is to act as a launcher for the Minesweeper game. When the program is executed, the **main** method is called, and it creates an instance of the **Minesweeper** class, which initializes and starts the game.By separating the game initialization logic into a separate **Minesweeper** class, it promotes modular and organized code structure. It allows for easier maintenance and scalability, as the game-related logic is encapsulated in a dedicated class.

[Hanafi Satriyo Utomo Setiawan / 5025211195] Start From Here

1. countSurroundingMines()

```
private void countSurroundingMines() {
   for (int i = 0; i < 10; i++) {
     for (int j = 0; j < 10; j++) {
       if (!mines[i][j]) {
         int count = 0;
         if (i > 0 && mines[i - 1][j]) count++;
         if (i < 9 && mines[i + 1][j]) count++;
         if (j > 0 && mines[i][j - 1]) count++;
         if (j < 9 && mines[i][j + 1]) count++;
```

```
        if (i > 0 && j > 0 && mines[i - 1][j - 1]) count++;

        if (i < 9 && j < 9 && mines[i + 1][j + 1]) count++;

        if (i > 0 && j < 9 && mines[i - 1][j + 1]) count++;

        if (i < 9 && j > 0 && mines[i + 1][j - 1]) count++;

        surroundingMines[i][j] = count;

      }

    }

  }

}
```

countSurroundingMines() counts the number of mines surrounding each cell in the grid and stores the counts in the **surroundingMines** array. Here's how the code works:

- The method iterates over each cell in the grid using two nested loops. The outer loop iterates over the rows (**i**), and the inner loop iterates over the columns (**j**).
- For each cell, it checks if there is a mine present (**!mines[i][j]**). If there is no mine in the current cell, it proceeds to count the number of mines in the surrounding cells.
- It initializes a variable **count** to 0, which will hold the count of surrounding mines for the current cell.
- The code then checks the neighboring cells to see if there are mines present. It uses a series of conditional statements to check the eight neighboring cells. Here's a breakdown of the conditions:
  - **if (i > 0 && mines[i - 1][j]) count++;** : Checks the cell above the current cell (if it exists) and increments count if there is a mine.
  - **if (i < 9 && mines[i + 1][j]) count++;** : Checks the cell below the current cell (if it exists) and increments count if there is a mine.
  - **if (j > 0 && mines[i][j - 1]) count++;** : Checks the cell to the left of the current cell (if it exists) and increments count if there is a mine.
  - **if (j < 9 && mines[i][j + 1]) count++;** : Checks the cell to the right of the current cell (if it exists) and increments count if there is a mine.

- ○ **if (i > 0 && j > 0 && mines[i - 1][j - 1]) count++;** : Checks the cell in the top-left diagonal direction (if it exists) and increments count if there is a mine.
  - ○ **if (i < 9 && j < 9 && mines[i + 1][j + 1]) count++;** : Checks the cell in the bottom-right diagonal direction (if it exists) and increments count if there is a mine.
  - ○ **if (i > 0 && j < 9 && mines[i - 1][j + 1]) count++;** : Checks the cell in the top-right diagonal direction (if it exists) and increments count if there is a mine.
  - ○ **if (i < 9 && j > 0 && mines[i + 1][j - 1]) count++;** : Checks the cell in the bottom-left diagonal direction (if it exists) and increments count if there is a mine.
- Finally, the code assigns the value of **count** to the corresponding element in the **surroundingMines** array. This array will store the count of surrounding mines for each cell.

2. uncoverCell()

```
private void uncoverCell(int i, int j) {
  if (mines[i][j]) {
    loseGame();
  } else {
    buttons[i][j].setText(Integer.toString(surroundingMines[i][j]));
    buttons[i][j].setEnabled(false);
    uncoveredCells++;
    if (uncoveredCells == 90) {
      winGame();
    }
    if (surroundingMines[i][j] == 0) {
      uncoverSurroundingCellsDFS(i, j);
    }
    }
}
```

uncoverCell() is used to handle the uncovering of a cell in the game. It takes two parameters, **i** and **j**, which represent the row and column indices of the cell to be uncovered. Here's how the code works:

- The method first checks if the cell contains a mine by accessing the **mines** array at the specified indices (**mines[i][j]**). If there is a mine in the cell, the **loseGame()** function is called. This means the player has uncovered a mine and has lost the game.
- If there is no mine in the cell, the code executes the **else** block. It performs the following actions:
    - Sets the text of the button in the **buttons** array at the specified indices to the string representation of the number of surrounding mines (**Integer.toString(surroundingMines[i][j])**). This displays the number of surrounding mines on the button.
    - Disables the button by calling **setEnabled(false)**. This prevents the player from interacting with the button further.
    - Increments the **uncoveredCells** counter. This keeps track of the number of cells that have been uncovered.
    - Checks if the number of uncovered cells is equal to 90. If so, it means that all non-mine cells have been uncovered, and the player has won the game. In this case, the **winGame()** function is called.
    - Checks if the number of surrounding mines for the current cell is 0. If it is, it means that there are no mines around the cell, and it is safe to automatically uncover the surrounding cells. The **uncoverSurroundingCellsDFS(i, j)** function is called to perform a depth-first search and uncover the surrounding cells.

3. uncoverSurroundingCellsDFS()

```
private void uncoverSurroundingCellsDFS(int i, int j) {
```

```
Deque<Point> stack = new ArrayDeque<>();

stack.push(new Point(i, j));

visited[i][j] = true;


while (!stack.isEmpty()) {

  Point point = stack.pop();

  int x = point.x;

  int y = point.y;


  if (surroundingMines[x][y] == 0) {

    if (x > 0 && !visited[x - 1][y]) {

      uncoverCell(x - 1, y);

      visited[x - 1][y] = true;

      if (surroundingMines[x - 1][y] == 0) {

        stack.push(new Point(x - 1, y));

      }

    }

    if (x < 9 && !visited[x + 1][y]) {

      uncoverCell(x + 1, y);

      visited[x + 1][y] = true;

      if (surroundingMines[x + 1][y] == 0) {

        stack.push(new Point(x + 1, y));

      }

    }

    if (y > 0 && !visited[x][y - 1]) {

      uncoverCell(x, y - 1);

      visited[x][y - 1] = true;

      if (surroundingMines[x][y - 1] == 0) {

        stack.push(new Point(x, y - 1));

      }

    }
    if (y < 9 && !visited[x][y + 1]) {
      uncoverCell(x, y + 1);
      visited[x][y + 1] = true;
      if (surroundingMines[x][y + 1] == 0) {
        stack.push(new Point(x, y + 1));
      }
    }
    if (x > 0 && y > 0 && !visited[x - 1][y - 1]) {
      uncoverCell(x - 1, y - 1);
      visited[x - 1][y - 1] = true;
      if (surroundingMines[x - 1][y - 1] == 0) {
        stack.push(new Point(x - 1, y - 1));
      }
    }
    if (x < 9 && y < 9 && !visited[x + 1][y + 1]) {
      uncoverCell(x + 1, y + 1);
      visited[x + 1][y + 1] = true;
      if (surroundingMines[x + 1][y + 1] == 0) {
        stack.push(new Point(x + 1, y + 1));
      }
    }
    if (x > 0 && y < 9 && !visited[x - 1][y + 1]) {
      uncoverCell(x - 1, y + 1);
      visited[x - 1][y + 1] = true;
      if (surroundingMines[x - 1][y + 1] == 0) {
        stack.push(new Point(x - 1, y + 1));
      }
    }
    if (x < 9 && y > 0 && !visited[x + 1][y - 1]) {
      uncoverCell(x + 1, y - 1);
      visited[x + 1][y - 1] = true;
```

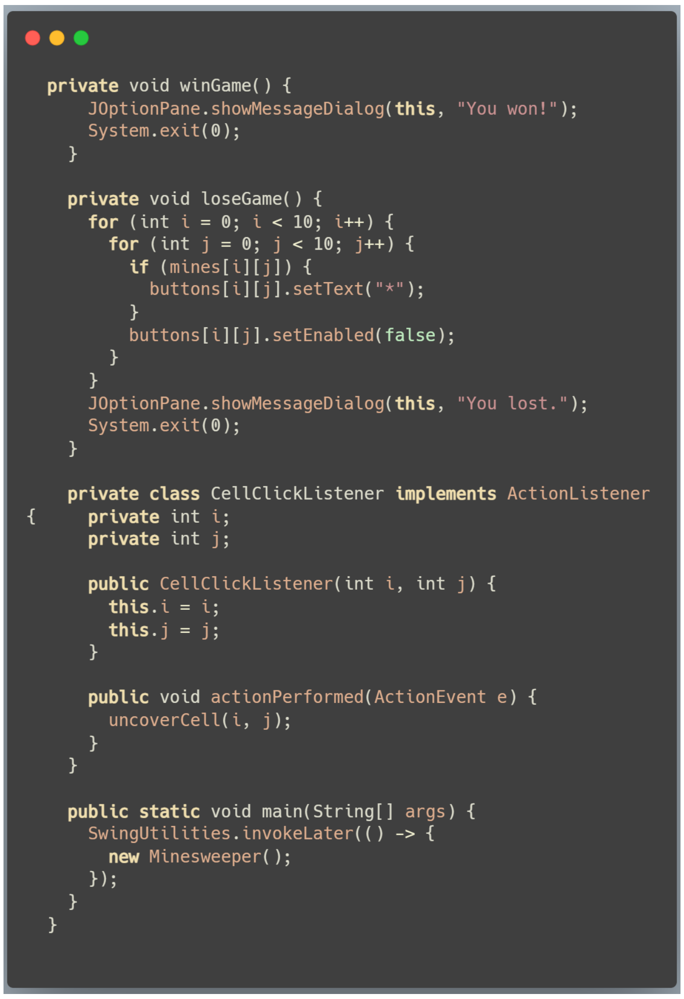
```
          if (surroundingMines[x + 1][y - 1] == 0) {

            stack.push(new Point(x +  1, y - 1));

          }

        }

      }

    }

}
```

uncoverSurroundingCellsDFS() is a depth-first search (DFS) algorithm. It is used to automatically uncover cells in the game that have no surrounding mines (cells with a value of 0 in the **surroundingMines** array). Here's how the code works:

- It starts by creating a stack (**Deque<Point> stack**) to keep track of the cells to be explored. It also initializes the stack with the current cell (**stack.push(new Point(i, j))**) and marks the current cell as visited (**visited[i][j] = true**).
- The code enters a **while** loop that continues until the stack is empty. This loop performs the **DFS traversal** of the grid.
- Inside the loop, it pops the top element from the stack (**Point point = stack.pop()**) and extracts its coordinates (**int x = point.x and int y = point.y**).
- It checks if the current cell has no surrounding mines (**surroundingMines[x][y] == 0**). If it does not have any mines around it, it proceeds to uncover the neighboring cells.
- The code checks the neighboring cells in eight directions and performs the following actions for each neighboring cell:
  - Checks if the neighboring cell is within the grid bounds and has not been visited (**!visited[x][y]**).
  - Uncovers the neighboring cell by calling the **uncoverCell()** method with its coordinates (**uncoverCell(x, y)**).
  - Marks the neighboring cell as visited (**visited[x][y] = true**).
  - If the neighboring cell also has no surrounding mines (value of 0 in **surroundingMines**), it pushes the cell's coordinates to the stack (**stack.push(new Point(x, y))**). This ensures that the algorithm will explore the surrounding cells of the current neighboring cell.

- The loop continues until the stack is empty. It explores and uncovers all cells with no surrounding mines and their neighboring cells recursively.

[Christian Kevin Emor / 5025211153] Start From Here



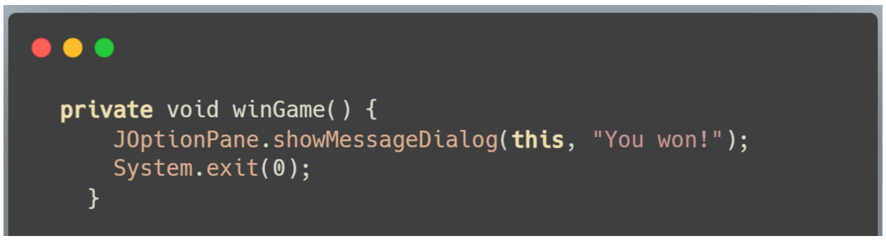
```
  private void winGame() {
      JOptionPane.showMessageDialog(this, "You won!");
      System.exit(0);
    }

    private void loseGame() {
      for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10; j++) {
          if (mines[i][j]) {
            buttons[i][j].setText("*");
          }
          buttons[i][j].setEnabled(false);
        }
      }
      JOptionPane.showMessageDialog(this, "You lost.");
      System.exit(0);
    }

    private class CellClickListener implements ActionListener
{     private int i;
      private int j;

      public CellClickListener(int i, int j) {
        this.i = i;
        this.j = j;
      }

      public void actionPerformed(ActionEvent e) {
        uncoverCell(i, j);
      }
    }

    public static void main(String[] args) {
      SwingUtilities.invokeLater(() -> {
        new Minesweeper();
      });
    }
  }
```

The provided code snippet contains the following functions:

1. winGame(): This function is called when the player wins the game. It displays a message dialog using JOptionPane to notify the player that they have won. Then, it terminates the program by calling System.exit(0), exiting the application.
2. loseGame(): This function is called when the player loses the game. It iterates over all the buttons on the game grid and checks if there is a mine (mines[i][j]). If a mine is found at a specific button, it sets the button's text to "*" to indicate a mine. Additionally, it disables all the buttons on the grid. It displays a message dialog using

JOptionPane to inform the player that they have lost. Finally, it terminates the program by calling System.exit(0).

3. CellClickListener class: This is an inner class that implements the ActionListener interface. It handles the action performed when a button on the game grid is clicked. The class stores the coordinates of the clicked button (i and j) in its instance variables. When the actionPerformed method is called, it invokes the uncoverCell(i, j) method to uncover the clicked cell.
4. main method: This is the entry point of the program. It creates an instance of the Minesweeper class using new Minesweeper(). The SwingUtilities.invokeLater() method is used to ensure that the GUI is created and executed on the Event Dispatch Thread (EDT), which is the thread responsible for handling Swing events and user interactions.



```java
private void winGame() {
    JOptionPane.showMessageDialog(this, "You won!");
    System.exit(0);
  }
```

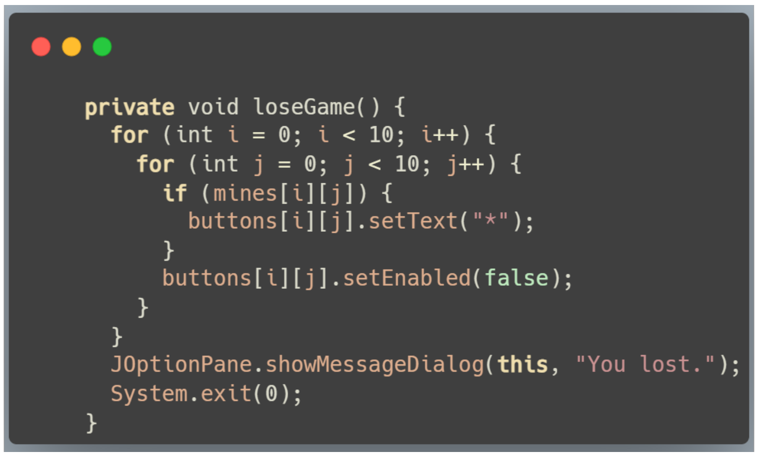
coding explanation:

1. JOptionPane.showMessageDialog(this, "You won!");

   This line displays a message dialog box using the JOptionPane class. The dialog box will show the message "You won!" to the user. The this keyword refers to the current instance of the class where this code is written.

2. System.exit(0);

   This line terminates the Java Virtual Machine (JVM) and stops the execution of the program. The argument 0 passed to the exit method indicates a normal termination of the program. This effectively closes the application and ends the game when the "You won!" message is displayed.



```
private void loseGame() {
  for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
      if (mines[i][j]) {
        buttons[i][j].setText("*");
      }
      buttons[i][j].setEnabled(false);
    }
  }
  JOptionPane.showMessageDialog(this, "You lost.");
  System.exit(0);
}
```

coding explanation:

1. for (int i = 0; i < 10; i++) {

   This line starts a for loop that initializes the variable i to 0. The loop will continue executing as long as i is less than 10. After each iteration, the value of i is incremented by 1.

2. for (int j = 0; j < 10; j++) {

   This line starts another nested for loop that initializes the variable j to 0. Similar to the previous loop, this loop will continue executing as long as j is less than 10. After each iteration, the value of j is incremented by 1.

3. if (mines[i][j]) {

   This line checks if the mines array at index [i][j] is true, indicating the presence of a mine at that position.

4. buttons[i][j].setText("*");

   If there is a mine at the current position [i][j], this line sets the text of the corresponding buttons array element to "*". This is typically done to indicate that the button represents a mine.

5. buttons[i][j].setEnabled(false);

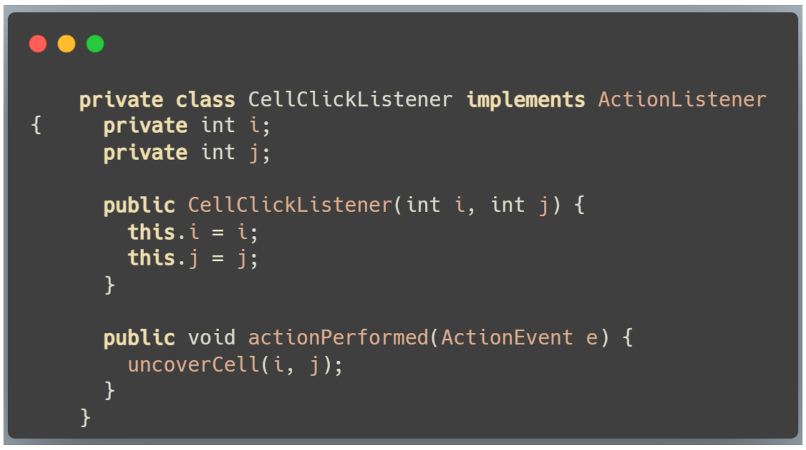
Regardless of whether there is a mine or not, this line disables the button at the current position [i][j] by setting its enabled property to false. This prevents further interaction with the button.

6. JOptionPane.showMessageDialog(this, "You lost.");

   This line displays a message dialog box using the JOptionPane class. The dialog box will show the message "You lost." to the user. The this keyword refers to the current instance of the class where this code is written.

7. System.exit(0);

   This line terminates the Java Virtual Machine (JVM) and stops the execution of the program. The argument 0 passed to the exit method indicates a normal termination of the program. This effectively closes the application and ends the game when the "You lost." message is displayed.



```
    private class CellClickListener implements ActionListener
{     private int i;
      private int j;

      public CellClickListener(int i, int j) {
        this.i = i;
        this.j = j;
      }

      public void actionPerformed(ActionEvent e) {
        uncoverCell(i, j);
      }
    }
```

coding explanation:

1. private int i; private int j;

   These lines declare two private integer variables i and j within the CellClickListener class. These variables are used to store the coordinates of a cell.

2. public CellClickListener(int i, int j) { this.i = i; this.j = j; }

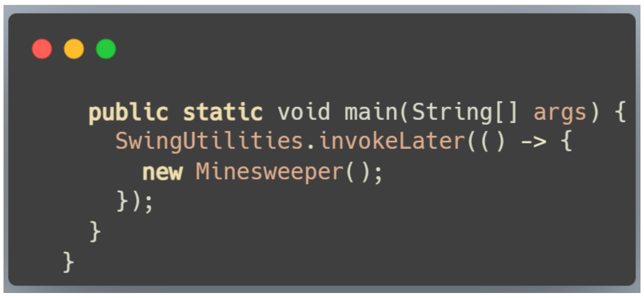
   This is the constructor of the CellClickListener class. It takes two integer parameters i and j. When an instance of CellClickListener is created, these parameters are used to initialize the i and j variables of that instance. The this.i and this.j refer to the i and j variables defined in the class. The this keyword is used to distinguish the instance variables from the constructor parameters with the same names. In other words, the constructor allows you to set the i and j values for each instance of the CellClickListener class.

3. public void actionPerformed(ActionEvent e) {

   This line declares the method actionPerformed with a parameter of type ActionEvent named e. The ActionEvent represents the action that occurred, such as a button click.

4. uncoverCell(i, j);

   This line calls the uncoverCell method and passes the i and j values as arguments. It is assumed that the uncoverCell method exists elsewhere in the code, and it is responsible for performing an action related to uncovering a cell in a grid or table. By calling this method with the i and j coordinates, the actionPerformed method triggers the uncovering action for the specific cell associated with this CellClickListener instance.



```
    public static void main(String[] args) {
      SwingUtilities.invokeLater(() -> {
        new Minesweeper();
      });
    }
  }
```

code explanation:

1. public static void main(String[] args) {

   This line declares the main method, which is the entry point of the Java program. It is the method that is executed when the program starts. The String[] args parameter allows command-line arguments to be passed to the program, although it is not used in this specific code snippet.

2. SwingUtilities.invokeLater(() -> {

This line calls the invokeLater method from the SwingUtilities class. This method is used to ensure that the code inside the provided lambda expression is executed on the event dispatch thread (EDT) in Swing applications. The EDT is responsible for handling user interface events and updating the UI components.

3. The () -> { new Minesweeper(); }

   is a lambda expression representing a Runnable object. It specifies the code to be executed on the EDT.

4. new Minesweeper();

   This line creates a new instance of the Minesweeper class. It invokes the default constructor of the Minesweeper class to initialize the object. Presumably, the Minesweeper class is a class that represents the game or application itself and contains the necessary logic and user interface components.

The Source Code:

Minesweeper.java

```
package com.minesweeper;

import java.awt.*;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.util.ArrayDeque;

import java.util.Deque;

import java.util.Random;

import javax.swing.*;


public class Minesweeper extends JFrame {

  private JButton[][] buttons;

  private boolean[][] mines;

  private int[][] surroundingMines;
```

```java
private boolean[][] visited;

private int uncoveredCells;


public Minesweeper() {

  setTitle("Minesweeper");

  setDefaultCloseOperation(EXIT_ON_CLOSE);

  setLayout(new GridLayout(10, 10));


  buttons = new JButton[10][10];

  mines = new boolean[10][10];

  surroundingMines = new int[10][10];

  visited = new boolean[10][10];

  uncoveredCells = 0;


  for (int i = 0; i < 10; i++) {

    for (int j = 0; j < 10; j++) {

      buttons[i][j] = new JButton();

      buttons[i][j].addActionListener(new CellClickListener(i, j));

      add(buttons[i][j]);

    }

  }


  placeMines();

  countSurroundingMines();


  pack();

  setVisible(true);

}
```

```java
private void placeMines() {
  Random random = new Random();
  int placedMines = 0;
  while (placedMines < 10) {
    int i = random.nextInt(10);
    int j = random.nextInt(10);
    if (!mines[i][j]) {
      mines[i][j] = true;
      placedMines++;
    }
  }
}

private void countSurroundingMines() {
  for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
      if (!mines[i][j]) {
        int count = 0;
        if (i > 0 && mines[i - 1][j]) count++;
        if (i < 9 && mines[i + 1][j]) count++;
        if (j > 0 && mines[i][j - 1]) count++;
        if (j < 9 && mines[i][j + 1]) count++;
        if (i > 0 && j > 0 && mines[i - 1][j - 1]) count++;
        if (i < 9 && j < 9 && mines[i + 1][j + 1]) count++;
        if (i > 0 && j < 9 && mines[i - 1][j + 1]) count++;
        if (i < 9 && j > 0 && mines[i + 1][j - 1]) count++;
        surroundingMines[i][j] = count;
      }
    }
  }
}

private void uncoverCell(int i, int j) {
  if (mines[i][j]) {
    loseGame();
  } else {
    buttons[i][j].setText(Integer.toString(surroundingMines[i][j]));
    buttons[i][j].setEnabled(false);
    uncoveredCells++;
    if (uncoveredCells == 90) {
      winGame();
    }
    if (surroundingMines[i][j] == 0) {
      uncoverSurroundingCellsDFS(i, j);
    }
  }
}

private void uncoverSurroundingCellsDFS(int i, int j) {
  Deque<Point> stack = new ArrayDeque<>();
  stack.push(new Point(i, j));
  visited[i][j] = true;

  while (!stack.isEmpty()) {
    Point point = stack.pop();
    int x = point.x;
    int y = point.y;
```

```
      if (surroundingMines[x][y] == 0) {
        if (x > 0 && !visited[x - 1][y]) {
          uncoverCell(x - 1, y);
          visited[x - 1][y] = true;
          if (surroundingMines[x - 1][y] == 0) {
            stack.push(new Point(x - 1, y));
          }
        }
        if (x < 9 && !visited[x + 1][y]) {
          uncoverCell(x + 1, y);
          visited[x + 1][y] = true;
          if (surroundingMines[x + 1][y] == 0) {
            stack.push(new Point(x + 1, y));
          }
        }
        if (y > 0 && !visited[x][y - 1]) {
          uncoverCell(x, y - 1);
          visited[x][y - 1] = true;
          if (surroundingMines[x][y - 1] == 0) {
            stack.push(new Point(x, y - 1));
          }
        }
        if (y < 9 && !visited[x][y + 1]) {
          uncoverCell(x, y + 1);
          visited[x][y + 1] = true;
          if (surroundingMines[x][y + 1] == 0) {
            stack.push(new Point(x, y + 1));
          }
        }
```

```
    if (x > 0 && y > 0 && !visited[x - 1][y - 1]) {
      uncoverCell(x - 1, y - 1);
      visited[x - 1][y - 1] = true;
      if (surroundingMines[x - 1][y - 1] == 0) {
        stack.push(new Point(x - 1, y - 1));
      }
    }
    if (x < 9 && y < 9 && !visited[x + 1][y + 1]) {
      uncoverCell(x + 1, y + 1);
      visited[x + 1][y + 1] = true;
      if (surroundingMines[x + 1][y + 1] == 0) {
        stack.push(new Point(x + 1, y + 1));
      }
    }
    if (x > 0 && y < 9 && !visited[x - 1][y + 1]) {
      uncoverCell(x - 1, y + 1);
      visited[x - 1][y + 1] = true;
      if (surroundingMines[x - 1][y + 1] == 0) {
        stack.push(new Point(x - 1, y + 1));
      }
    }
    if (x < 9 && y > 0 && !visited[x + 1][y - 1]) {
      uncoverCell(x + 1, y - 1);
      visited[x + 1][y - 1] = true;
      if (surroundingMines[x + 1][y - 1] == 0) {
        stack.push(new Point(x +  1, y - 1));
      }
    }
  }
```

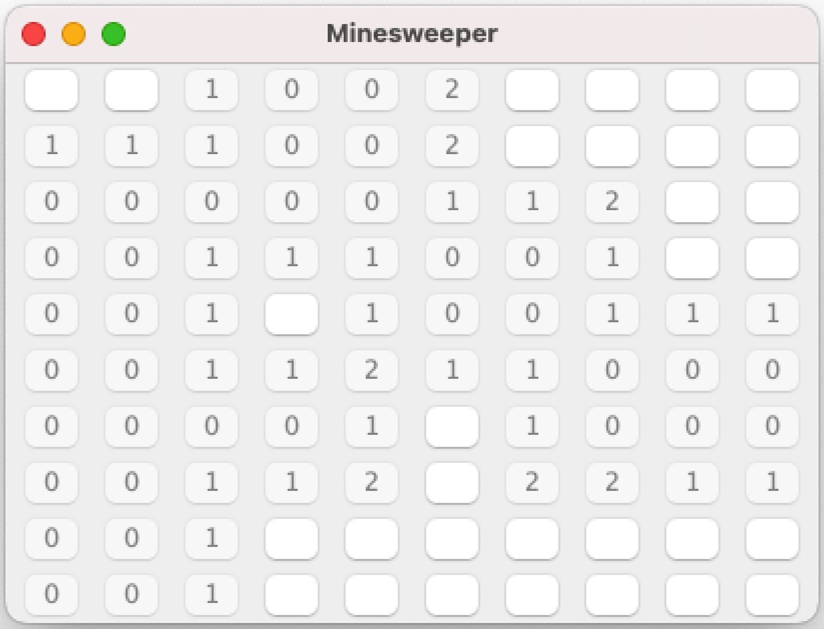
```java
    }
  }
   private void winGame() {
     JOptionPane.showMessageDialog(this, "You won!");
     System.exit(0);
   }

   private void loseGame() {
     for (int i = 0; i < 10; i++) {
       for (int j = 0; j < 10; j++) {
         if (mines[i][j]) {
           buttons[i][j].setText("*");
         }
         buttons[i][j].setEnabled(false);
       }
     }
     JOptionPane.showMessageDialog(this, "You lost.");
     System.exit(0);
   }

   private class CellClickListener implements ActionListener {
     private int i;
     private int j;

     public CellClickListener(int i, int j) {
       this.i = i;
       this.j = j;
     }
```

```
         public void actionPerformed(ActionEvent e) {
            uncoverCell(i, j);
         }
      }

      public static void main(String[] args) {
         SwingUtilities.invokeLater(() -> {
            new Minesweeper();
         });
      }
   }
```

Main.java

```
package com.minesweeper;

public class Main {
      public static void main(String[] args) {
            new Minesweeper();
      }
}
```

Output Program:



```
    1 0 0 2
1 1 1 0 0 2
0 0 0 0 0 1 1 2
0 0 1 1 1 0 0 1
0 0 1   1 0 0 1 1 1
0 0 1 1 2 1 1 0 0 0
0 0 0 0 1   1 0 0 0
0 0 1 1 2   2 2 1 1
0 0 1
0 0 1
```

Conclusion

In conclusion, our team succeeded in developing a Minesweeper game that incorporates various algorithms taught in our lectures. The game provides an interactive and engaging experience for players while demonstrating the application of key algorithms such as Depth-First Search (DFS) and randomization.During the implementation process, we faced the challenge of managing game state and ensuring optimal performance. However, by leveraging our understanding of the DFS algorithm and leveraging randomization techniques, we managed to overcome these challenges and achieve a game that is both functional and fun.

Game evaluation and analysis revealed satisfactory performance in terms of uncovering cells and dealing with game logic. The DFS algorithm efficiently finds neighboring cells without mines in the vicinity, minimizing the amount of user interaction required. Additionally, the randomization algorithm facilitates fair placement of mines, adding an element of unpredictability to every game session.Overall, this project gave us hands-on experience in applying algorithmic concepts to real-world problems. This strengthens our programming skills, fosters collaboration within our team, and deepens our understanding of algorithms and their practical applications.

We are grateful for the opportunity to work on this project and the knowledge gained during its development. It served as a valuable learning experience, allowing us to bridge the gap between theoretical knowledge and practical implementation.

By the name of Allah (God) Almighty, herewith I pledge and truly declare that I have solved quiz 2 by myself, did not do any cheating by any means, did not do any plagiarism, and did not accept anybody's help by any means. I am going to accept all of the consequences by any means if it has proven that I have done any cheating and/or plagiarism.

Surabaya, 18 Mei 2023

[illegible]	[illegible]	[illegible]
[M. Armand Giovani]	[Christian Kevin Emor]	[Hanafi Satriyo Utomo S.]
[5025211054]	[5025211153]	[5025211195]