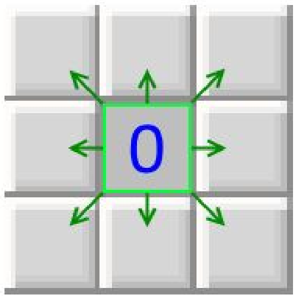
2a. We have coded the game Minesweeper in Java, using LWJGL, Lightweight Java Game Library. Our code also includes 3 different game modes with 1 gamemode having 2 power ups and 2 debuffs. Our program was created to reduce stress and provide entertainment. The video illustrates the process of starting our game and the playing of a gamemode. The game mode we choose to demonstrate is the game mode Arcade. The Arcade gamemode is the only game mode that uses power ups and debuffs, such as Bullseye, Sonar, Amnesia, and BombSpread.

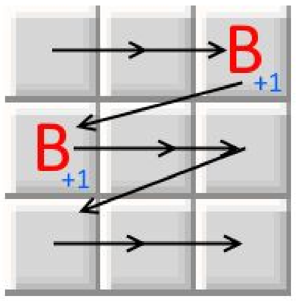
2b. Given that we had two weeks to code an entire program and do the written responses we choose a game that was older. We had narrowed down our search to Minesweeper, Tetris, and Snake. Tetris would take the longest out of the 3 while, Snake would be the shortest. Even though Snake would take the shortest it was also the most duplicated out of the 3, so we chose Minesweeper. One problem with coding a game that's been coded a million times is how to make it unique. So in order for us to be unique we made use of 2 power ups and 2 debuffs with 3 different game modes. Another difficulty was the trouble of getting the code from each other and from the school to our homes. We overcame this obstacle by uploading our code to Github. By uploading our code to Github we good update the code and download the code with ease. Since we were creating our game from scratch, we had to create our own sprites and by doing so we were able to use our artistic abilities. The development was mainly collaborative since we were helping each other solve problems or fix an error in code. There are parts in the code where only one of us coded, two examples being the game modes and the interface.

```java
public void recursiveCheck(int x, int y) {
    if((level[x][y].Number == -1) && (level[x][y].Checked == false)) {
        if(getNearbyBombs(x, y) == 0) {
            level[x][y].Number = 0;
            level[x][y].Checked = true;
        try {
            recursiveCheck(x - 1, y - 1);
            recursiveCheck(x, y - 1);
            recursiveCheck(x + 1, y - 1);
            recursiveCheck(x - 1, y);
            recursiveCheck(x, y);
            recursiveCheck(x + 1, y);
            recursiveCheck(x - 1, y + 1);
            recursiveCheck(x, y + 1);
            recursiveCheck(x + 1, y + 1);
            } catch(Exception e) {

            }
        } else {
            level[x][y].Number = getNearbyBombs(x, y);
            level[x][y].Checked = true;
            if(mode == GameMode.ARCADE) {
                randomQuestion(x, y);
            }
        }
    }
}
```



```java
public int getNearbyBombs(int x, int y) {
    int BombAmount = 0;
    for(int xi = (x - 1); xi < (x + 2); xi++) {
        for(int yi = (y - 1); yi < (y + 2); yi++) {
            if(xi < 0 || xi >= level.length || yi < 0 || yi >= level[0].length) {
                // Out of Bounds, Do Nothing
            } else {
                if(level[xi][yi].Bomb == true) {
                    BombAmount++;
                }
            }
        }
    }
    return BombAmount;
}
```



Total Bombs: 2

```java
public Minesweeper() {
    createWindow();
    TileImport.Import();
    Mouse.setGrabbed(true);
    menu = new MainMenu();
    gameMenu = new GameMenu();
    gameLoop();
}

public void gameLoop() {
    while(!Display.isCloseRequested()) {
        MouseListener.tick();
        KeyboardListener.tick();

        renderUpdate();

        Display.update();
        Display.sync(60);
    }
}
```

2c. The code represented in part one of the image above has two of the major algorithms used to calculate the math and logic in the board. The recursiveCheck() method is a recursive method that calls upon itself continuously until all possible situations that fit the methods conditions are fulfilled. The *rescursiveCheck()* without the recursion is a simple method which checks to see if the tile has not been touched, and that there are no bombs nearby the tile. If in the end, there are no bombs nearby then the tile becomes an empty tile. Otherwise, we use the method *getNearbyBombs()* to check if there is an X amount of bombs nearby. Then that tile on the board, represents X, the number of bombs that are nearby. Now that all the tiles have been given a designated type they are then "checked" meaning they cannot be changed by the player. The *getNearbyBombs()* method simply adds up all the bombs around a tile with a 1 tile radius using a for loop and an if statement to check a 3x3 area.

2d. The second part of the image above contains the code that is ran at the start of the game as well as the game loop. Inside the initializer there is multiple examples of abstraction like the method *createWindow()* and *TileImport.Import()* these both are used to reduce the complexity and size of the program by organizing each bit of code to its own method of what that code does. For example, *TileImport.Import()* would be getting all the colors needed to draw something, while *createWindow()* would create the canvas to draw on. The *gameMenu* object which contains the gamemodes that the player can choose from, sends the gamemode into a new object instance of *Game* class. Then the *Game* class takes the parameters and uses it to load a new game of the user's choice. Another example of abstraction could include the *getNearbyBombs()* method. This method simplifies the code by counting the number of bombs within an area. It is used numerous times in the games code as well as in *resursiveCheck()* without the *getNearbyBombs()* method it would make the code more complex and well as extensively long having to calculate the number of bombs with 5 or 10 lines when instead one line can be repetitive and annoying to look at when coding. Infact, without methods to reduce simplicity most of java would just be an endless amount of random lines with no guide to help understand what exactly is happening in the program at a given time.