# CS 455 Programming Assignment 3

Spring 2020 [Bono]
**Due:** Wed., March 25, 11:59pm

---

## Introduction

In this program you will get a chance to use recursion to solve a problem that could not be done just as easily or more efficiently with a loop. It is possible to solve this problem without recursion, but it would somewhat more complicated, and would result in the same big-O space and time requirements as a recursive solution.

Depth-first search (DFS) is an algorithm for searching in non-linear data structures. There are a few different ways to implement DFS, but here you're going to implement a recursive DFS as part of a Minesweeper game. We'll discuss more about how to do this recursive search later.

The code for this assignment uses a few features of Java we haven't used in this course before. They are: 2D arrays and inner classes. Two-dimensional arrays were covered in section 7.6 of the textbook. The other features listed here are part of the starter code we gave you, and you will not need to write code using them yourself to complete the assignment. They are: inner classes in section 10.5, anonymous inner classes in Special Topic 10.4, and their use for Listeners (part of the Java GUI system) in section 10.7.2. Event handling in general is covered in Section 10.7.

Because this program has a GUI, the Vocareum environment is configured so you can open up a Linux desktop, like we had for pa1, so you can test the complete program

## The assignment files

Note: the blurbs below do not completely describe what each of these classes are, or how they fit together. For more details on that, see the section on the class design.

The starter files we are providing for you on Vocareum are listed here. The files in `bold` below are ones you create and/or modify and submit. The files are:

- `VisibleField.java` Data representing what's visible about the minefield and other data and methods for the game. The interface has been specified for you; you need to complete the implementation of this class.
- `MineField.java` The hidden minefield. The interface has been specified for you; you need to complete the implementation of this class.
- `MineSweeper.java` The main program. We have written this class for you. Do not change it.
- `MineSweeperFixed.java` A version of the program that uses a hard-coded mine field instead of randomly placed mines. (For testing purposes.) You may modify this file (e.g., to add more test mine field configurations) or not; we won't be grading it.
- `GameBoardPanel.java` The GUI for the game. We have written this class for you. Do not change it.
- `images/facesmile.gif` Image used by the program.

- `images/facedead.gif` Image used by the program.
- `MineFieldTester.java` and `VisibleFieldTester.java` Test programs you may create to test your two classes. You are not required to submit these files. These are described further in the section on Development Hints.
- `README` See section on Summary of requirements for what to put in it. Before you start the assignment please read the following statement which you will be "signing" in the README:

> "I certify that the work submitted for this assignment does not violate USC's student conduct code. In particular, the work is my own, not a collaboration, and does not involve code created by other people, with the exception of the resources explicitly mentioned in the CS 455 Course Syllabus. And I did not share my solution or parts of it with other students in the course."

## The assignment

Write the main logic of a program to play the game Minesweeper. To keep the scope of this assignment reasonable, we have created the object-oriented design for this program, and have implemented the GUI for you.

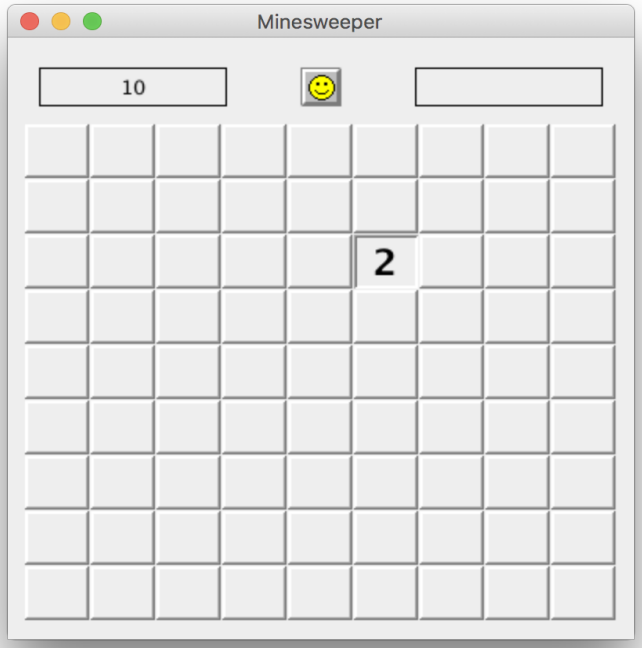
Minesweeper is a game where mines are hidden in various random locations in a two-dimensional minefield. The player has to figure out where all the mines are without exploding a mine. They do this by repeatedly uncovering locations, and guessing mine locations, until they uncover all of the *non*-mine locations (win) or they explode a mine (lose). This is more than a game of chance, because when a non-mine square is uncovered it displays the number of mines adjacent to that square. You can use that information to figure out where the mines are (or at least narrow down their locations).

The rest of this section describes the game in somewhat gory detail. You do not need to know *all* of the details (e.g., of exactly how the display changes) to actually play the game yourself. And the first part of your homework is to go out and play several games of minesweeper to get an idea of how it works. If you have never played before, you probably want to start out with beginner mode (9 x 9 field, 10 mines). There are many versions of this game. It's been on Windows forever, and probably is still. Here is a free web-based version. There are also a few free smart-phone minesweeper apps. Just don't get hooked, like I did, or you'll never get to your assignment!

Note: The user interface for our implementation is a little more simplistic than in the real game. For example for our game the player can't change the size of the minefield or number of mines for a game. (You would have to change the program itself to do that.)

**Uncovering a square.** The player uncovers a square with a left-click on a button at that location. If it's non-mine location, what shows up at that square is the number of adjacent hidden mines (a number 1 - 8 because diagonals are considered adjacent). If there are no mines adjacent to this square, it displays no number. If the player uncovers a mine instead, it explodes, and they lose the game.

---

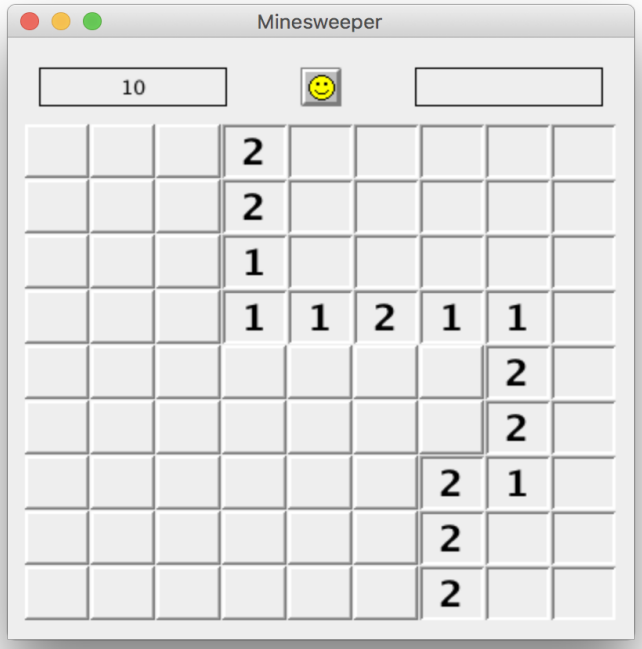



A game after player opens the first square. It's showing that two of the squares surrounding that square have mines in them. Our game has a 9 x 9 field, with 10 mines.

---

**Automatic opening of empty regions.** To make it less tedious for the user, when they open a square that has no adjacent mines, the game automatically opens all the squares in that region that aren't adjacent to mines until it gets to the boundary of the field, or squares that are adjacent to other mines. Here is an example that illustrates the effect:

---




The player only clicked once, at the location: 2nd row, 7th column. It wasn't adjacent to any mines, so the program automatically opened the rest of the squares shown.

---

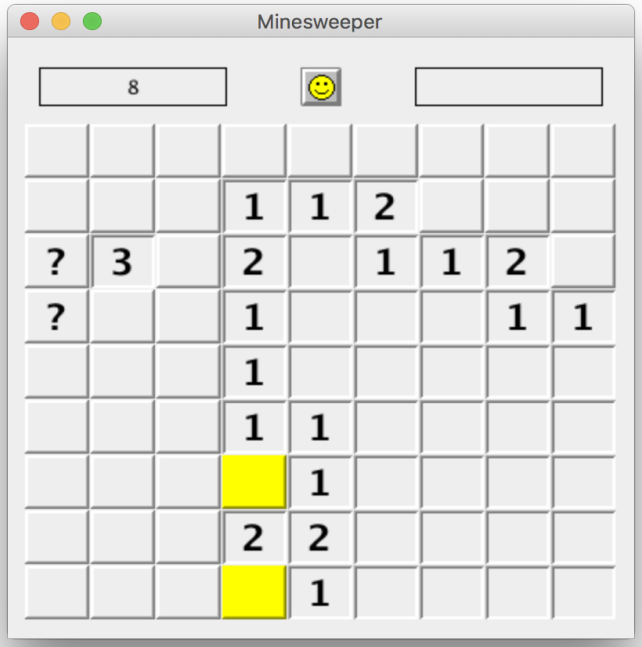
(This automatic opening of parts of the field is where your recursive search is going to come into play.)

**Guessing a mine location.** Based on the opened squares, if the player can then deduce the location of a mine, they can mark that unopened location as a mine-guess (right-click), to help them figure out where more mines are and are not. (In our interface, the mine guess is denoted by a yellow square.)

**Marking what may be a mine location.** Usually a player would guess a mine location only when they could actually deduce that a mine goes there. However, there is a third state a covered mine location can have, and that's to just mark it with a question mark, to show that it's something they are thinking might be a mine, but aren't sure about. Marking a location this way does not affect the number of mine guesses.

If a mine is covered, one right click puts it as a mine guess (yellow), and another right click puts it as a question (shows ?, not yellow), and a third right click puts it back to the regular covered state.

---




This player has made a few guesses (yellow) and marked a few spots they are unsure of with a question mark. The large empty area (shown as pushed in buttons), was automatically opened by the program when the player earlier clicked on just one of those empty spots. The top left shows the number of mines left to guess.

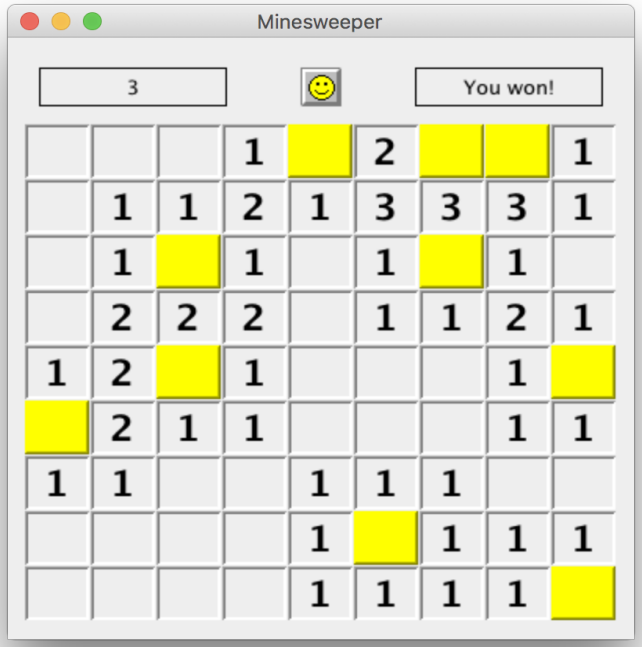
---

**Number of mines left display.** When the game starts the number of mines still to be guessed is displayed at the upper left of the window, so the user knows how close they are to finishing. Guessing a mine (i.e., marking that location, as described above), will reduce that number by one *whether the guess was correct or not*. Although a user can guess more mines than there actually are, the display only goes down to zero (no negative number displayed). Note: some versions of the game out there let the displayed number go to negative, but our version will not.

The user can win without it reaching zero, because ***the win-condition is to open all the non-mine locations***, not to guess any particular number of mines. The next diagram illustrates this situation.

**Winning game display.** When a player successfully opens all of the non-mine locations, the field display changes to show where the other mines are (it shows them as guesses, in yellow). I.e., these would be any unopened squares that weren't already yellow. The top middle of the window will show the happy face, and a message about winning will appear on the upper right of the window.

---

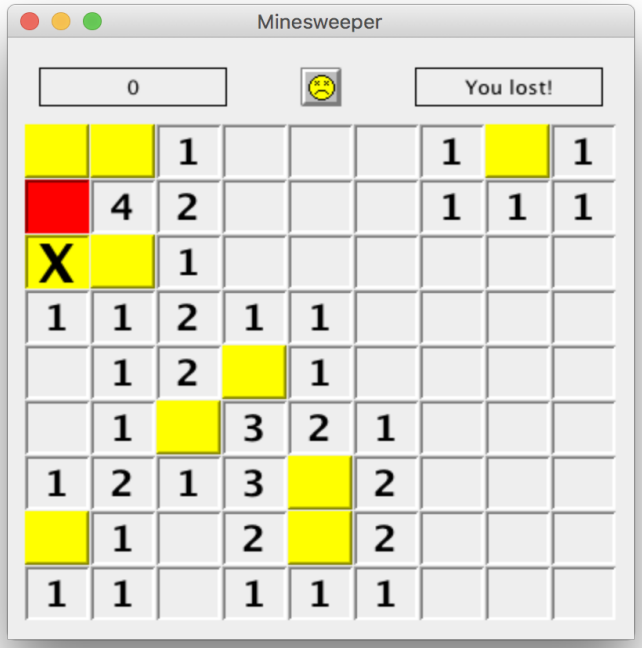



This player just won the game (indicated by "You won!" message at upper right) . The number of guesses made was 7 (upper left display shows 10 - 7 = 3), but once they opened all the 71 (9*9 - 10) non-mine squares they won. It shows 10 yellow squares, because on a win, all of the squares with mines are shown as guesses.

---

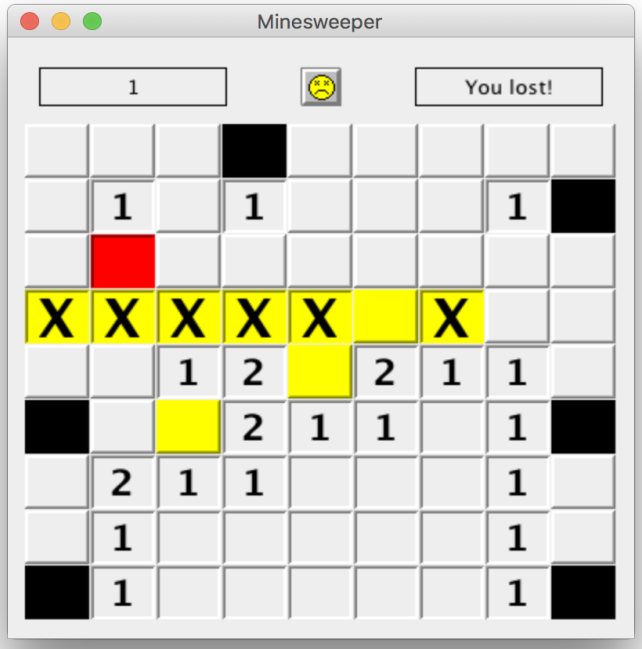
**Losing game display.** When a player opens a mine location, that mine explodes, and they lose the game. The exploded mine is shown by a red square. Any previously made incorrect guesses are shown with an X though them, the correctly guessed mine locations are still shown as guesses (yellow), and the other unopened mines are shown as "mines" (a black square, in our implementation). The top middle of the window will show the sad face, and a message about losing will appear on the upper right of the window.

---




This player just lost the game at the very end. They guessed 9 squares correctly, the last one incorrectly, and then opened the last unopened square, which was a mine. The red one is the mine they clicked on, the X is the incorrect guess.

---




Another game just lost. The black squares are mines that the player hadn't yet guessed at the point that they lost. The X's are incorrect guesses. The's 3 other yellow ones were the correct guesses. The red square is the exploded mine.

---

**Playing the game again.** The user can play another game by clicking on the happy/sad face. It goes back to a happy face at the start of the new game, the game board shows all

squares as unopened, and shows the original number of mines in the display of how many left to guess (top left).

**Initial mine placement.** On each game played, the mine locations are chosen at random, thus will be different for each game. The game works such that the first square opened is guaranteed not be a mine. To get that behavior the program can't choose the mine locations until the user opens the first square.

**Clicking on an already-opened square.** In our game clicking on a square that has already been opened will do nothing -- this is different than some other versions of the game you can play. (In other versions of the game clicking on an open square that has a number on it is a shortcut for opening all the neighboring non-guessed squares.)

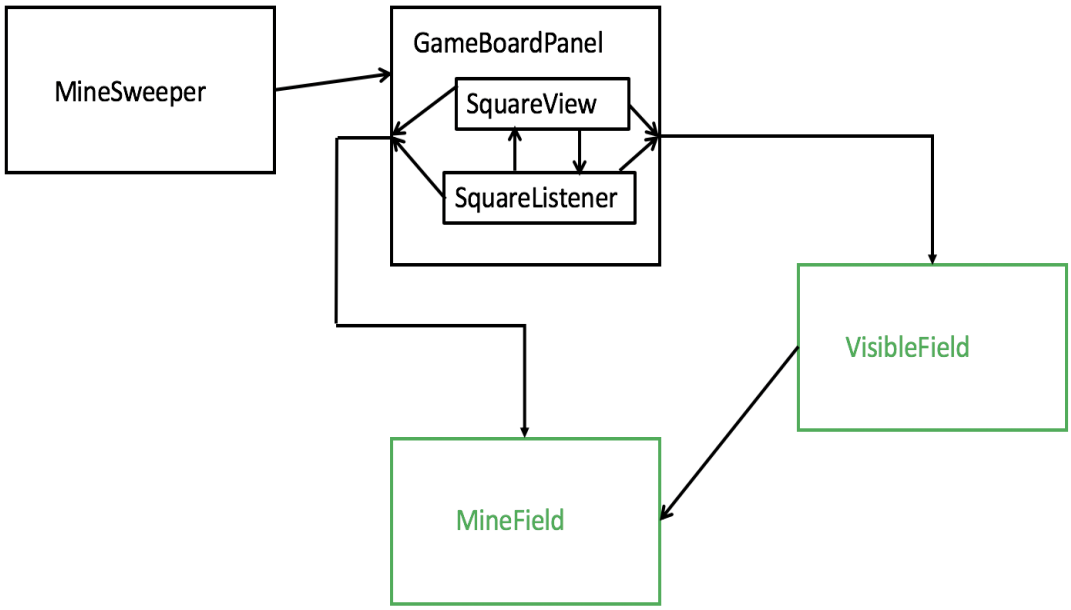
We are giving you the overall design to use, as well a lot of the code, so you should read on before you dive into this problem.

## The Class Design

We have done the class design for you. Normally in a GUI program it's desirable to separate the part of the program that does the computation and stores data from the part of the program dealing with the display and input. This organization is called the Model-View-Controller design pattern. This way it's easier to change the look and feel of the program without having to touch the code that does the computation: e.g., we could even use our `MineField` and `VisibleField` classes, below, unchanged, with a non-graphical, console-based interface. The design given here has that separation. The part you are implementing, `MineField` and `VisibleField`, comprises the Model.

All of the programming assignments you have had so far have had a version of this design pattern. In PA1 the computation happened in `CoinTossSimulator`, and display of the computation was in `CoinSimComponent` and the `Bar` class used by `CoinSimComponent`. In PA2, the `BookshelfKeeper`, and `Bookshelf` classes handled the data and computation, and all I/O (including error checking) was handled in `BookshelfKeeperProg`.

The simple UML (Unified Modeling Language) class diagram below shows these relationships between the classes in our design for the minesweeper program. An arrow from class A to class B means that class A depends on class B (or put other ways, "has to know about", or "uses"). To keep it simpler, we showed the inner classes as contained inside their enclosing class. The classes in green are the ones you are implementing for the assignment.




Here is an overview of this design, stating which classes are responsible for which part of the program, and what the dependencies are between the various classes:

- `GameBoardPanel` This `JPanel` subclass contains all the display elements and listeners in the program. It has a two-dimensional array of `SquareView` objects for all of the buttons on the minefield board. During the normal mode of operation, it creates the `MineField` and `VisibleField` objects. (So it depends on those other two classes.) This class has already been completed for you.

  This is the GUI for the program: it contains the display and controls for a game, and the minefield display (grid of "buttons"). It's the View and Controller in the MVC design pattern, whereas the Model consists of the the `VisibleField` and `MineField`. You do not need to understand every line of code in `GameBoardPanel` to complete this assignment; a lot of it is minutiae of setting up and updating Java Swing components. However, it will be helpful if you know when and why the various methods you are implementing get called from methods in this class and its inner classes. So here's more about its design. It has two named inner classes:

  - `SquareView`. The view class for a single square. There is one of these objects for each square on the board. Although it looks like a button, this is actually a JLabel, because it's easier to use a JLabel to respond to both left and right clicks. It also uses your `VisibleField` accessors to find out the state of the square it's displaying. We wrote this class for you.

  - `SquareListener`. The controller class for a single square. There is one for each square on the board. Its one public method, `mouseClicked`, detects if it was a left or right click, and dispatches it to its private method `openSquare` (left click), or `changeGuessStatus` (right click). This class depends on the `SquareView`, `GameBoardPanel`, `MineField`, and the `VisibleField`. We wrote this class for you. Listeners are described in a little more detail below.

  While there is a view and controller for each individual square, the design has no separate "Square" class to model a square. The state of all of the squares is in the

`VisibleBoard` class. The view and controller for a square know their own location; they call `VisibleBoard` methods, passing in their location to specify their square.

The listener for the other button on the board (the smiley face, to start a new game) is an anonymous inner class. That means it doesn't have a name, because it's just used in one place. It's in the `setupTopPanel` method of the `GameBoardPanel` class.

- `MineSweeper` class. This class has `main`. It sets up the frame and `GameBoardPanel`. It depends on the `GameBoardPanel`.

- `MineField` class. Has locations of the mines for a game. This is the "hidden" part of the minefield. This class is mutable, because we sometimes need to change it once it's created. Its mutators are:
  - `populateMineField`. The first time the user clicks on a square on the board, this is called to place the random hidden mines, preventing a mine from going into the just-clicked location.
  - `resetEmpty`. Put the minefield into its starting state (before that first click).

  This class does no I/O. Doesn't depend on any other classes.

- `VisibleField` class. This is the data that's being displayed at any one point in the game (i.e., visible field, because it's what the user can see about the minefield). It does no drawing on the screen itself. Client can call `getStatus(row, col)` for any square to find out the current display status. It actually has data about the whole current state of the game, including the underlying mine field (`getMineField()`). Other accessors related to game status: `numMinesLeft()`, `isGameOver()`. It also has mutators related to moves the player could do (`resetGameDisplay()`, `cycleGuess()`, `uncover()`), and changes the game state accordingly.

  It, along with the `MineField` (accessible in `mineField` instance variable), forms the Model for the game application, whereas `GameBoardPanel` is the View and Controller, in the MVC design pattern. That `MineField` can be accessed (or modified) from outside this class via the `getMineField` accessor.

  This class does no I/O. Depends on the `MineField` class.

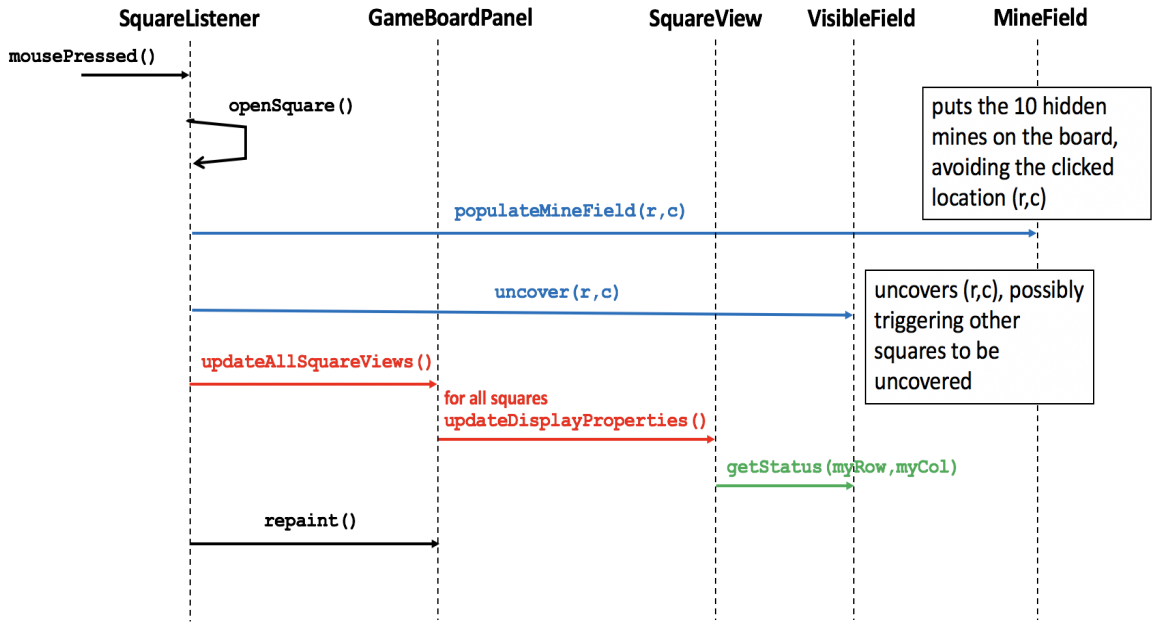
---

**What's a listener?**A listener is a GUI component that has a function that gets called automatically whenever a certain user action happens (e.g., a mouse click or a key pressed). It's usually associated with a particular component: e.g., the component might be a button, and the listener has the code that gets executed when that button is pressed). The application programmer fills in the body of this function for a subclass of the appropriate Java listener class.

---

## A sample use-case of `MineSweeper`

Let's go through what happens with the classes for a particular action. The following is another type of UML diagram, called a sequence diagram, that shows the message-passing (aka, method calls) between objects to carry out a particular action. You read the diagram from top to bottom. The use-case being described in this sequence diagram is that the user left-clicks a minefield square for the first time in a game. The last thing that happens in this sequence is the window being updated, with the call to `repaint`).




Use case: user opens the first square.

We can relate this specific diagram to the general idea of communication between classes in the MVC pattern:

- The **blue** steps shows the Controller updating the Model
- The **red** steps shows the Controller telling the View that the Model changed.
- The **green** steps show the View asking the Model for its updated state.

And the parts in the black boxes on the right side of the diagram are annotations describing what happens in the Model (i.e., the part you are implementing).

## Algorithm hints

There are two parts of solving this problem that may be challenging. We'll give you some hints on how to solve those part here.

### Choosing the initial mine locations

You can use a straightforward technique to choose random mine locations. That is, choose a random location, and if that location already has a mine on it, keep choosing another random location until you get a location that doesn't have a mine on it. Because of a precondition on the 3-parameter `MineField` constructor, that the number of mines has to be less than one-third of the total number of possible mine locations, there is a very low probability that choosing locations this way will result in very many false tries.

### Recursively finding an open area

You are required to use recursion in your solution to the problem of finding an open area with no mines in it. As mentioned earlier, you will be using a variation on depth-first search to do this. More specifically, you will be using the flood-fill algorithm, which is used in graphics for completely filling in an enclosed area with a particular color, pixel-by-pixel (which itself is a specialization of the algorithm for finding a connected component in a graph). The pseudocode for flood-fill in this Wikipedia article should give you a good start

for figuring out how to solve this problem applied to minesweeper. The parts of the article you will need are from the start of the section "The Algorithm" through "Stack-based recursive implementation" (the "stack" mentioned is not explicitly created, but is the underlying call stack used in all recursion). The algorithm presented there assumes 4-way connectivity, while, of course, here we have 8-way connectivity.

## Development Hints

You are encouraged to write private helper methods to make your code clearer. (That's true for any program for this course.) In particular here, the recursive search part of your code will almost certainly be a private helper function.

Here is a possible plan for how you might develop your code:

1. Get the 1-arg constructor and all accessors for the `MineField` class working before you write or test the other methods. You may also, optionally, add a `toString` method to help you see that your fields got initialized properly (the only change to the MineField interface that we will allow). Your tests should be in a `MineFieldTester` class you create, that creates several `MineField` objects whose data come from hard-coded arrays, and calls the accessors on them to figure out whether they were created correctly.

2. You could then proceed to implement a subset of your `VisibleField` class, so you can test it on mine-fields with known mine locations using a `VisibleFieldTester` you create, and in a more limited way (but easier to see results) with `MineSweeperFixed`, a main program we provided for you that uses a hard-coded minefield. You can also modify `MineSweeperFixed` to use different fixed minefields of your own creation).

   A subset you could create here is to make a game that doesn't automatically open the empty areas (doesn't implement the recursive search part), but requires the user to open each square manually.

3. Once you are sure this version of `VisibleField` works you could add the recursive search feature to it, and retest with a modified `VisibleFieldTester` and with `MineSweeperFixed`.

4. Now you can add the `MineField` functionality for random mine placement, and retest on a modified `VisibleFieldTester` and with `MineSweeper`.

## Summary of requirements

As on the previous assignment, there are several requirements for this assignment related to design, testing, and development process strewn throughout this document. We'll summarize those and the functional requirements here:

- implement `MineField` and `VisibleBoard` classes according their public interfaces. This includes not adding any new public methods (besides toString), or adding/changing/removing any public named constants. You are welcome to add any necessary private named constants or private helper functions.
- `MineField` and `VisibleBoard` must work correctly with the rest of the given minesweeper code.

- use recursion to implement the automatic opening of empty squares. (a solution that does not use recursion will receive little credit)
- your code will also be evaluated on style, documentation, and design. We will deduct points for programs that do not follow all of the published style guidelines for this course (they are also linked from the Assignments page).
- add necessary information to the `README` file. A reminder of what goes in it: This is the place to document known bugs in your program. That means you should describe thoroughly any test cases that fail for the the program you are submitting. (You do not need to include a history of the bugs you already fixed.) You also use the README to give the grader any other special information, such as if there is some special way to compile or run your program. You will also be signing the certification shown near the top of this document.

**A note about grading of this assignment**

Because the code *you* are writing has no I/O, much of the grading of the assignment will involve auto-grading such that we submit your `MineField` and `VisibleMineField` classes to our own unit-tests. So **you will need to spend some effort unit-testing these classes yourself: making sure that they work to specification, not only that they happen to work with the given `MineSweeper` program.**

Furthermore, to be able to compare your results to predictable expected results, many of our tests will use the non-random version of your code (i.e., using the 1-parameter `MineField` constructor, the one that takes a 2D array). *If this constructor doesn't work for your program, you will lose a significant amount of credit on this assignment, because these tests will fail.*

Similar to other unit-tests we have discussed in this class, the structure of many of these test cases we will subject your code to are:

- call a mutator (or constructor)
- test that the object is in the correct state from that mutator by:
  calling all of the accessors, testing that actual results match the expected results

If you have not done such tests yourself, you might not be sure that the code works correctly. This is a more thorough and systematic test than just running the game a few times on arbitrary input.

For more about developing your program, see the Development Hints section.

## Submitting your program

You will be submitting completed versions of `MineField.java`, `VisibleField.java`, and `README`. Make sure your name and loginid appear at the start of each file. Reminder: if you developed your code locally on your laptop, before submitting you will need to upload your code to Vocareum, and compile and re-test it using the Linux Desktop on Vocareum before submitting it.

Please try submitting the first time at least a day before the final submission deadline so you have time to fix your program based on any failed tests.

When you click the "Submit" button in your PA3 Vocareum work area, it will check that you have the correct files in your work area and whether they compile. It will also do a few tests on your MineField class: (1) it will test whether your MineField has a correct 1-arg constructor and whether the accessors return correct values for a MineField created with that constructor, (2) it tests the 1-argument constructor with a non-square minefield, and (3) tests that that constructor does a defensive copy of its argument. Please remember to check the submission report.

*Passing* these submit checks is not necessary or sufficient to submit your code (the graders will get a copy of what you submitted either way). (It would be necessary but not sufficient for getting full credit.) However, if your final submitted version does not pass all the tests we would expect that you would add some explanation of that in your README and resubmit with the updated README. One situation where it might fail would be if you only completed a subset of the assignment (and your README would document what subset you completed.)

You are allowed to submit as many times as you like, but we will only grade the last one submitted. If you are unsure of whether you submitted the right version, there's a way to view the contents of your last submit in Vocareum after the fact: see the item in the file list on the left called "Latest Submission".

---