

Project 5: MineWalker

CS121 Computer Science I

Objectives

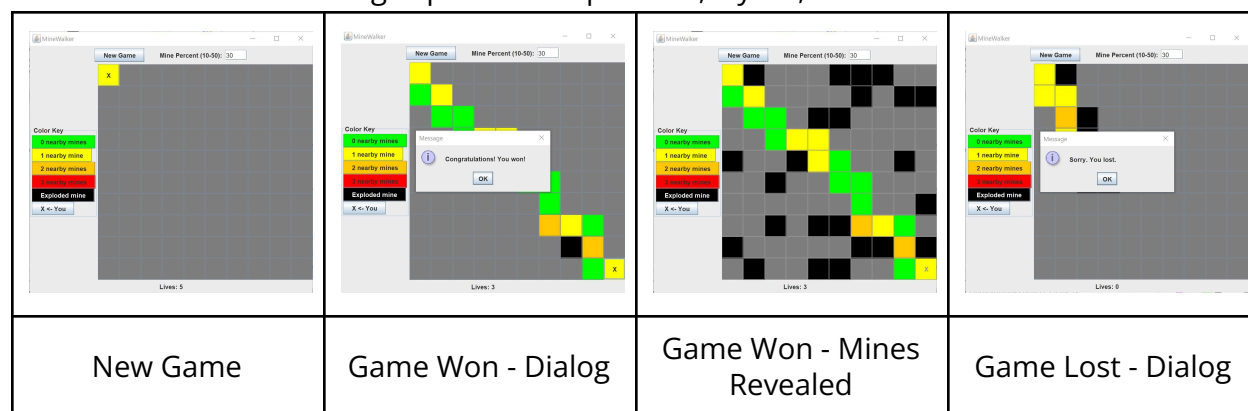
- Develop GUI application based upon event-driven programming paradigm
- Design a good user interface
- Use existing classes
- Use the Java API to learn how to properly use unfamiliar classes
- Use inheritance to extend the functionality of a class

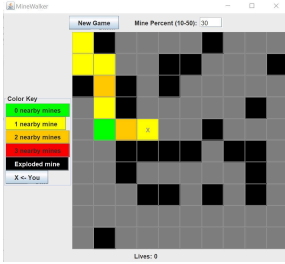
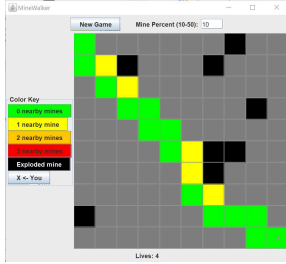
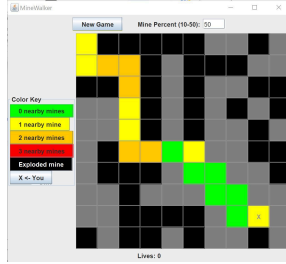

Overview

MineWalker is a game in which the player attempts to navigate a hidden minefield from the top left corner of a grid to the lower right corner. Exposed tile colors indicate how many mines are in adjacent positions. If the player runs out of lives from stepping on mines before reaching the goal, the game is lost.

From the player's current position, they can only move up, down, left, or right within bounds. Stepping on a mine reveals the mine and costs the player one life. The player never actually enters a mine position - if a mine is exposed, the player remains in their previous position. Once a mine is exposed, the player cannot step on that mine a second time.

Screenshots demonstrating expected components, layout, and features:



			
Game Lost - Mines Revealed	10% Mines	50% Mines	Example Alternate Sub-Panel Layout

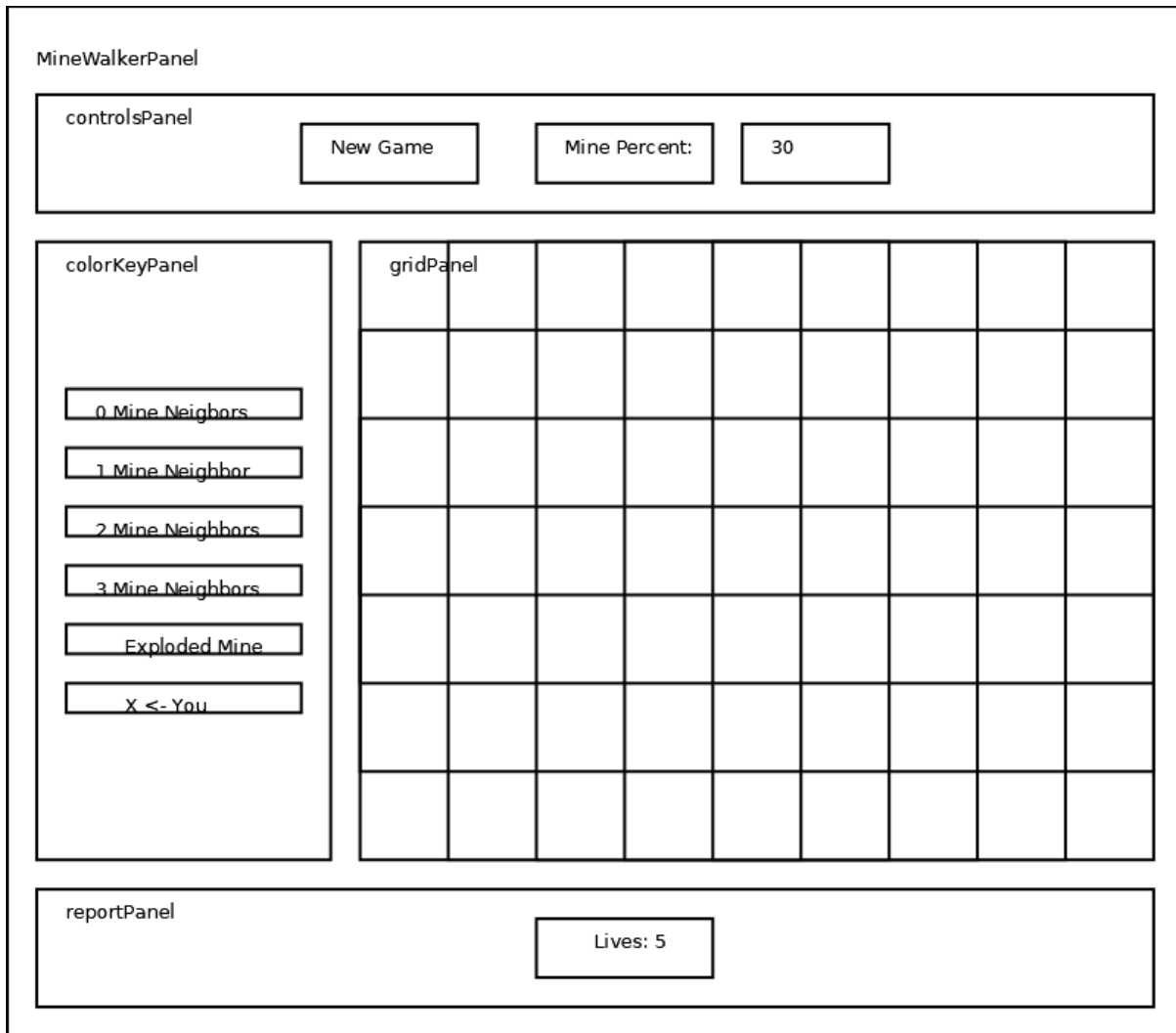
Getting Started

- Create a new Java project in VS Code named “MineWalker”
- Download the [project support files](#) and import them into the project.
 - [TileButton.java](#)
 - [RandomPath.java](#)
- Create a driver class named `MineWalker.java`. It will be responsible for creating the `JFrame` and instantiating the `MineWalkerPanel.java` class (described next).
- Create a class named `MineWalkerPanel` to serve as the top level panel for the GUI. Be sure to extend `JPanel` because `MineWalkerPanel` “is-a” `JPanel`.

Development Strategy

Before writing a single line of code, read **all** project specifications and sketch out the design of the program based on the following diagram, either on paper or a whiteboard. First label all components as `JButton`, `JLabel`, `TextField`, `JPanel`, etc.. Then identify the layout managers each panel will need to organize its sub-panels or other components. This will be incredibly helpful as you begin writing the code to lay out the GUI.





Begin coding by writing the `MineWalkerPanel` constructor to establish the layout and placement of all components in their sub-panels and sub-panels within `MineWalkerPanel`. Make sure all components are reasonably sized and positioned. Configure the game grid for a new game.

Develop the `ActionListener` for `TileButton` clicks where most of the game logic is defined. (Don't forget to add the listener to each `TileButton` where they are first created!)

Once a single game is working, develop the new game button `ActionListener` to reset the game grid for a new game (and add this listener to your new game button where it is created). Consider a new game method that can be called from this listener and also used in the constructor to configure the first game.



There are many useful and relevant examples of GUI code under the [chap06 examples](#). Especially recommended examples are [LayoutDemo](#), [BorderDemo](#), and [CoordinatorExample](#) and their associated classes. The official [Java Tutorials](#) and Java API are also full of useful guidance on using layouts and components.

Project Specification

Quality Requirements (20 points)

1. Project Submission
 - Code submitted to the correct cs121 section and with **only** the required files
2. README.md
 - README.md file is named properly and has the required sections including your reflection (see [Documentation](#) section)
3. Javadoc Comments
 - Project should have proper Javadoc formatted comments (see [Documentation](#) section)
4. Coding Style
 - Code should be properly indented and have reasonable and consistent vertical spacing
 - Variable and method names should follow the *camelCase* naming convention
 - Class names should follow the *TitleCase* naming convention
 - Constants should be in all *CAPITAL_LETTERS*
 - In-line comments to describe what different code sections do

Code Requirements (80 points)

The specifications for each class are listed below.

Class: RandomPath.java (given as project handout)

This static utility class has a single method that generates and returns a random path of Points connecting the upper left and lower right corners of a grid with given dimensions, where each step is closer to the destination than its predecessor. This could potentially be made more interesting, but leave it as-is for this project.

Class: TileButton (given as project handout)

This is a customized JButton that should be used for all tiles in the game board. TileButton has instance variables for its location, whether it is hidden, whether it is a mine, whether it is part of the clear path, and its number of adjacent neighbors that are mines. As a JButton, it can also have its color changed with setBackground(), have its



text changed with `setText()`, and be enabled or disabled with `setEnabled()`. Do not modify this class.

Class: MineWalker.java

This class is the driver class. It should create the `JFrame` with title `MineWalker` and containing `MineWalkerPanel`. All GUI layout and logic is delegated to the `MineWalkerPanel`.

Class: MineWalkerPanel.java

This is the top-level panel for the `MineWalker` GUI. It lays out all other components of the program and defines `ActionListeners` to update game state and start new games.

- In the constructor, create all components and sub-panels to organize them. Set the layout manager of the main panel to `BorderLayout` and create sub-panels for the game grid, color key, game controls, and game status information. Recommended layout for the controls panel is the default `FlowLayout` or horizontal `BoxLayout`. Recommended layout for the game status panel is default `FlowLayout`. Recommended layout for the color key panel is vertical `BoxLayout`. Set up the game grid for the first game so the player can start playing right away.
- The **only** instance variables in `MineWalkerPanel` should be those that are needed in listeners. These include a 2D array of `TileButtons`, the number of remaining lives, the `JLabel` used to display the number of remaining lives, the `TextField` for specifying the percentage of mines, and the `TileButton` corresponding to the player's current position. All other variables should be local to the constructor or methods where they are needed.
- There are many potentially useful constants that should be considered in `MineWalkerPanel` including number of starting lives, grid dimension, default mine percentage, colors corresponding to different game tile states, etc.
 - Game grid dimension should be 10 (for a 10x10 board).
 - For each game the player starts with 5 lives.
 - The default percent of mines in non-path tiles is 30 percent.
 - Color for a hidden tile is gray.
Color for an exposed mine is black.
Color for a tile with 0 mine neighbors is green.
Color for a tile with 1 mine neighbor is yellow.



Color for a tile with 2 mine neighbors is orange.

Color for a tile with 3 mine neighbors is red.

- When initializing the game grid, each `TileButton` should be initialized and added to both to a game grid sub-panel with a `10x10 GridLayout` and to a `10x10 2D TileButton` array such that tile locations in the array match their locations as displayed in the GUI. (The 2D array is necessary because it is not possible to access tiles by coordinates in a `GridLayout`.)
 - Initialize each button to be enabled, be hidden, not be on the path, not be a mine, have no mine neighbors, have no text, and be the default gray color. This is also when you will add an `ActionListener` for `TileButton` events to each tile.
 - Get a new path from `RandomPath`. For each `Point` in the path, set the `TileButton` with corresponding coordinates to be a path tile.
 - Calculate the number of mines that need to be placed. The number of free spaces on the grid is the number of tiles minus the length of the path. The number of mines to place, then, is:
$$\text{numMinesToPlace} = (\text{numTiles} - \text{pathLength}) * \text{minePercent} / 100;$$
Until all mines are placed, generate random row,col coordinates in the grid and attempt to place a mine there. If the `TileButton` at those coordinates is not on the path and is not already a mine, set it to be a mine and decrement the number of mines to be placed. (Note - this is a place where poor logic can cause an endless loop in your program if you try to place more mines than you have free tiles.)
 - After all mines are placed, walk through each tile in the grid to sum and set its number of neighboring mines. Look up (if not in the top row), down (if not in the bottom row), left (if not in the left column), and right (if not in the right column). Do not look diagonally.
 - Set the current player `TileButton` to the top left `TileButton` and set that tile's text to "X" to indicate the player's location. Expose that tile and set its color according to its number of neighboring mines. The game should now be ready to play.
- Since `MineWalkerPanel` is the top-level panel, this is a great place to implement the `ActionListeners` for `TileButtons` and the new game button because `ActionListeners` in `MineWalkerPanel` have access to the game grid, all buttons, and all labels that are also defined in `MineWalkerPanel`.



- Define a private inner class that implements the `ActionListener` interface to respond to `TileButton` clicks. (Do not forget to add a reference to this listener to all `TileButtons` where they are created.) Most of the game logic is contained in the `actionPerformed()` method of this listener, as follows:
 - If the clicked `TileButton` is not adjacent to the player's current `TileButton` (up, down, left, or right of the current `TileButton`), ignore the click.
 - If the clicked `TileButton` is an already-exposed mine, ignore the click.
 - If the clicked `TileButton` is a hidden mine, set its hidden property to false, change its color, and decrement the player's number of lives by one. Do not move the player. To prevent the player from clicking the same mine twice, it would be a good idea to disable this button for the remainder of the game. If you do this, you won't need to check if a clicked tile is an exposed mine, above.
 - If the clicked `TileButton` is a hidden tile, set the tile's hidden property to false, color the tile according to its number of neighboring mines, set the text of the current player `TileButton` to an empty `String`, update the player's current `TileButton` reference to the clicked `TileButton`, and set the new player `TileButton` text to "X" to identify the player's current position on the grid.
 - If the clicked `TileButton` is not hidden (and not a mine), simply update tile text and the current player `TileButton` reference to move the player. (This allows the player to backtrack through previously visited tiles looking for a safer route.)
 - After all of the above, if the number of lives has been reduced to zero, the game is lost. Use a `JOptionPane` `MessageDialog` to inform the player of their unfortunate situation.
 - After all of the above, if the player's current position is the lower-right corner position (max row, max column), the game is won. Use a `JOptionPane` `MessageDialog` to inform the player that they have emerged victorious.
 - If the game is over for either reason, disable all `TileButtons` in the grid to prevent further movement. Reveal the location of all hidden mines by changing their color to the exploded color (or another distinct color).
- Define a private inner class that implements the `ActionListener` interface to respond to your New Game button clicks. (Do not forget to add a reference to this listener to the New Game button where it is created.)



- Reset all tiles to be enabled, hidden, not a mine, not on the path, have no mine neighbors, have no text, and be the default gray color.
- Get a new path and update all path tiles to know they are on the path.
- Place mines in non-path, non-mine tiles. Get the percentage of mines to use from the JTextField in the controls sub-panel.
- Place the player in the starting upper left tile and reset the num lives.

Testing

The program should be tested continuously throughout the development process to quickly catch errors. Once the project is complete and ready for submission, copy your project code to a folder on onyx (if it is not already there) and make sure it compiles and runs correctly **on onyx** before submitting.

Check that code compiles on onyx

- `javac *.java` - There should be no compile errors or warnings

Manual GUI Testing

- `java MineWalker`

Check that the GUI runs and allows the user to start a game right away with default settings. Confirm that only valid moves are allowed. Confirm that winning and losing conditions are correctly recognized and reported. Confirm that each new game uses the configured percentage of mines. Confirm that no additional moves can be made after a game has ended until a new game is started.

Enhancements to attempt only after all requirements are met

These suggestions would not only make the program more interesting but also provide excellent opportunities to sharpen the coding skills you have developed in this course! If you want to maintain and improve your coding skills after this course, these are highly recommended challenges.

Add toggle buttons to show/hide all hidden mine tiles or path tiles. These could be noted by a color change or a text marker. (These features would be most useful for the developer as they basically make the game, itself, pointless if used by the player. Consider having these features enabled or disabled by a “development tools enabled” constant in the code.)



Add a "hint" button to reveal one hidden mine adjacent to the player's current position (if there is one) without exploding it. Consider reducing the player's score if they use this tool.

Add a scoring scheme. For example, the initial score can be five times the number of tiles in the grid (e.g. 500 for a 10x10 grid). Each life costs the user 100 points and each step costs them 1 point. The goal is to lose as few points as possible. Or invent your own scoring system! Consider taking the percentage of mines into account when calculating the maximum score.

Add a Top 10 scores list. Store top scores in a file so you can reload the history of top scores each time the program starts. Display top scores in a new side panel.

Make grid size configurable as another way to customize game difficulty. See this example of how to remove and replace a panel: [GameBoard.java](#), [GameBoardPanel.java](#)

Validate text field inputs. Reset out-of-range or invalid inputs to the default value or the previous valid value. Try [slider bars](#) as an alternative to text field inputs. Set the range of values and let the slider position determine the current value.

When revealing all mines after a game is over, visually distinguish between mines that were exploded and mines that remained hidden. For example, color hidden mines a dark gray rather than black.

Enhance the path-building algorithm in RandomPath to allow some up and left moves for more interesting and challenging paths. I recommend no more than 30% probability of an up or left move for each new step on the path. You will have to detect and prevent revisiting a point already in the path. Enabling such interesting paths introduces the possibility of dead-ending during path generation (no valid moves from the path's current endpoint), so you'll have to recognize this situation and restart the path from scratch until a complete path is found.

Replace the color coding scheme with icons displayed on exposed tiles.

Add sound effects!

Documentation

If you haven't already done so, add [javadoc comments](#) to each class and all methods.

- Javadoc *class* comments should include both a short description of the class AND an @author YOURNAME tag.
 - The class javadoc comment is placed before the class definition



- Your class comment must include the @author tag at the end of the comment. This will list you as the author of your software when you create your documentation.
- Javadoc *method* comments should be placed before ALL methods.
 - Methods that are defined by an interface do not require additional comments since they are provided within the interface specification.

Include a plain-text file called **README.md** that describes your program and how to use it. **It should also include your reflection on the project.** Expected formatting and content are described in [README_TEMPLATE.md](#). See [README_EXAMPLE.md](#) for an example.

