Assignment 3

CS 1027
Computer Science Fundamentals II

Due date: Tuesday, November 14 at 11:55 pm

Learning Outcomes

In this assignment, you will get practice with:

- Using an interface
- Implementing a Stack class
- Using stacks to track 2D movements
- Working with exceptions
- Writing algorithms
- Programming according to specifications

Introduction

You and your friends decided to procrastinate your midterm studying and instead, went out to explore an abandoned factory. While rummaging through an old, dark warehouse, you stepped on a wooden trapdoor that immediately broke and left you freefalling into a deep chasm. Thankfully, there was a large mound of dirt that broke your fall at the bottom so you were virtually unharmed other than some pain in your legs. You quickly noticed that you fell into an old mine shaft and it's so deep that lava is flowing all around you that will instantly kill you if you step on it. Your heart filled with deep regret of your poor decision to procrastinate your studying.

You need to find your way out of the mine and back to safety -- perhaps the ladder will be a safe way out, but it also might lead you to another mine shaft on the way up. This means you may have to find your way through several of these mines. There are some locked doors that prevent you from walking but you might find some keys left by the miners from centuries ago. You may also stumble upon some gold while down there so you should grab onto it. It may be worth something. But if you step anywhere near the lava, the gold will melt and you will lose it.

In this assignment, you will be given code to graphically represent the mines. Each mine is stored in a text file which you can load and it will be displayed graphically so you can see the mine and watch the movement through the mine once you implement the path-finding algorithm.

Each mine is divided into a grid of square cells. You can walk from one cell to another adjacent cell but you cannot step onto lava or walk through a wall. There are several different types of cells which are described and shown below. Some cells allow you to pick up an item such as a key or a piece of gold ore. A list of all the different types of cells is provided below in Figure 1.

From a cell in a mine, there will be up to 4 neighbour cells on which you may be able to walk. The neighbouring cells are indexes 0 for the north neighbour (above), 1 for the east neighbour (to the right), 2 for the south neighbour (below), and 3 for the west neighbour (to the left). Note that some cells will have only two or three neighbours if they are at a corner or along an edge. However, if they have less than four neighbours, the indexing is still the same. For example, the starting cell in Figure 2 has only three neighbours since it is along an edge, and its neighbours are at index 1, 2, and 3. The neighbour that would have been at index 0 does not exist so it has a "null" 0th neighbour. The other neighbours maintain their indexing (1 is still the east neighbour, and so on). It is important when writing your code that you remember to check if a neighbour is not null before trying to access information about the cell.

Cell Types

Floor Cell You can walk freely on floor cells.	
Start Cell You begin on the start cell and it behaves like a regular floor cell	
Exit Cell You must reach the exit cell to escape the mine.	
Wall Cell You cannot walk on a wall cell.	

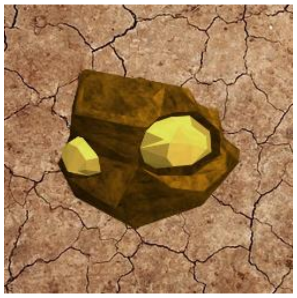
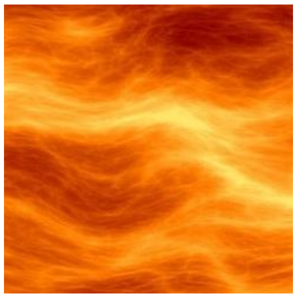
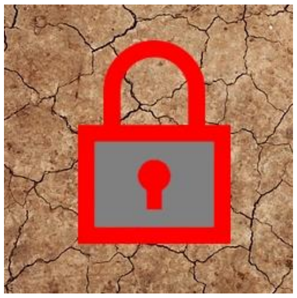
Gold Cell You may walk on a gold cell and you will pick up the gold.	
Lava Cell You cannot walk on a lava cell. You may walk adjacent to one but any gold you are holding will be destroyed and lost.	
Key Cells (red, green, and blue) You may walk on a key cell and you will pick up the key of that colour (pictured here is a blue key cell – note that there are also red and green key cells in which the key will be outlined in red or green, respectively).	
Locked Door Cells (red, green, and blue) You will need a key matching the colour of the locked door in order to unlock it and walk onto it. Without a key to unlock it, you cannot walk onto a locked door cell (pictured here is a red locked door cell – note that there are also green and blue locked door cells in which the lock will be outlined in blue or green, respectively).	

Figure 1. The list of all types of cells including the cell name, image, and a description of what they do and/or how they can be walked on.

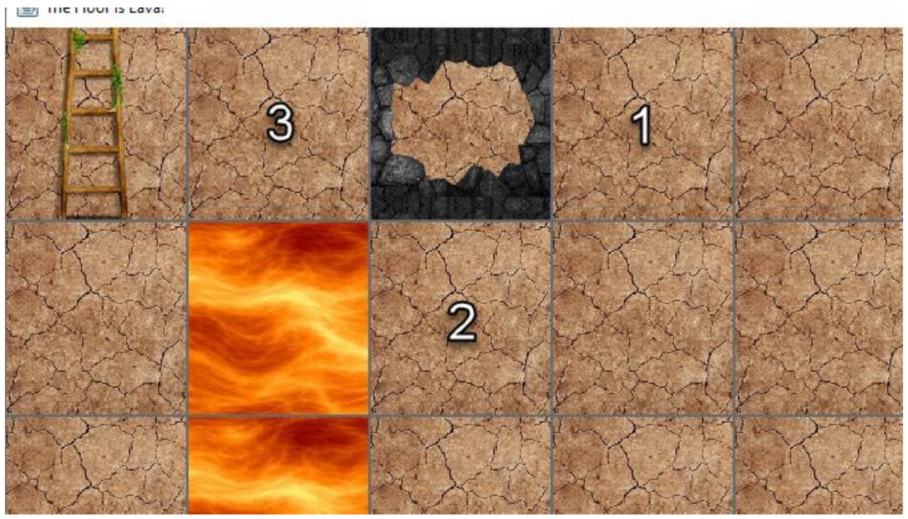


Figure 2. An illustration of the indices of neighbouring cells from a given cell (in this case, the starting cell) including one that is null (neighbour at index 0 in this example).

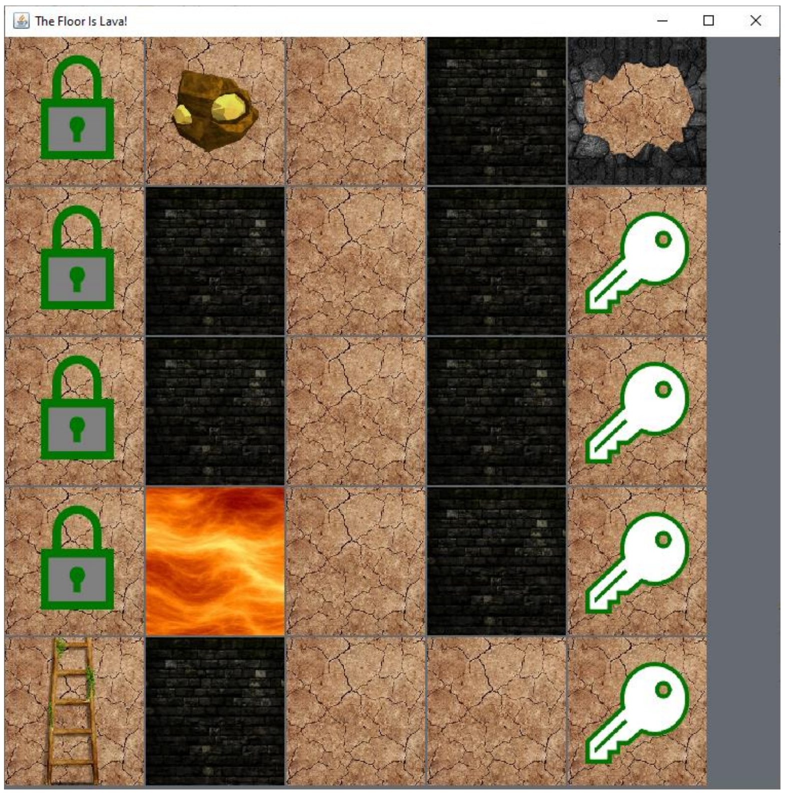


Figure 3. A depiction of a mine that includes a lava cell, four green locked door cells, four green key cells, and a gold cell.

An example of a mine is shown in Figure 3. In this mine, the starting cell is in the upper right corner and the exit cell is in the lower left corner. The mine has many walls and one lava cell, none of which can be walked on, so there's only one walkable path through this mine. There are four green locked door cells before arriving at the exit cell, but thankfully there are four green key cells on the way over there. The mine also has a gold cell which you will pick up on your way out, but unfortunately it will not last because you will have to walk adjacent to a lava cell which will melt the gold and you will lose hold of it.

Movement Rules and Restrictions

Your task is to use an algorithm to find your way out of a mine. This algorithm must use a stack to keep track of unexplored locations in the mine (given below as part of the MineEscape class). Part of this larger algorithm is a smaller algorithm that defines how to move from one cell to the next cell in the mine, given below. Since there may be several possible cells to move to from any cell, we want to apply a consistent algorithm, so that all solutions produce the same path to the exit on the same mine. Additionally, there are rules that refer to the markings on cells: as you walk through a mine, the cells you step on will have to be marked as being visited to ensure that you don't visit the same cell more than once.

The algorithm to find the next cell to explore must adhere to the rules and restrictions explained here. Failing to follow the rules exactly as specified will likely result in all/most the test cases failing. In this explanation, consider the term "curr" to represent the current cell on which you are already standing.

Choose the next cell to walk onto from curr such that:

- the next cell is not a wall and not a lava cell
- the next cell is not marked (i.e., it is not a cell we have already walked on)
- the next cell is decided based on the following set of ordered rules (i.e., follow the first rule that applies)
    1. if curr is adjacent to the exit cell, go to the exit cell
    2. if curr is adjacent to one or more cells that contain a collectible item (a key or gold), go to the neighbour with the smallest index containing a collectible
    3. if curr is adjacent to one or more floor cells, go to the neighbour with the smallest index that is a floor cell
    4. if curr is adjacent to one or more locked door cells, go to the neighbour with the smallest index that is a locked door cell for which you have a key of the same colour.
    5. if none of these conditions are met, return null to indicate that you cannot proceed and must backtrack

The role of this algorithm, as part of the overall mine escape algorithm, is presented in the section on the MineEscape class below.

Provided files

You will be provided with many classes for your project. Several of these classes will be needed to create the graphical user interface and some are classes that you will have to use while implementing your code. Here is a list of the provided classes along with some notable methods you may need to use from these classes:

- **CellComponent**
- **CellLayout**
- **IllegalArgumentException**
- **InvalidMapException**
- **InvalidNeighbourIndexException**
- **Map**
    - getStart()
- **MapCell**
    - changeToFloor()
    - getID()
    - getNeighbour(int i)
    - isMarked(), isMarkedInStack(), isMarkedOutStack()
    - markInStack(), markOutStack()
    - isStart(), isExit(), isFloor(), isWall(), isLava(), isGoldCell(), isLockCell(), isKeyCell(), isRed(), isGreen(), isBlue()
- **StackADT**
- **StackException**
- **TestStack**
- **TestPath**

The important methods from the MapCell class are described in the next section.

TestStack.java and TestPath.java are tester files to **help** check if your classes are implemented correctly.

Similar files will be incorporated into Gradescope's auto-grader. Additional tests will be run that will be hidden from you. **Passing all the tests within the provided files does not necessarily mean that your code is correct in all cases.**

You will also be given many images (.jpg) and text files (.txt). The images are used for the graphical representation of the mines and the text files represent the mines themselves (i.e. how each mine is comprised of the different cell types). For projects in Eclipse, all the images and text files should be **saved in the project's root directory – not in src nor in bin**. If you use a different IDE, you may have to look up online where to place the images and text files so that they work properly. It may be the same as Eclipse but some IDEs may require them to be saved in a different folder.

More Information About MapCell

The provided class MapCell is one of the foundational classes for this project. This represents the cells that will comprise the mines. There are many simple but important methods in this class that will help you when implementing your path-finding algorithm.

- Each MapCell object has a unique int ID. This can be obtained using getID(). While looking at the graphical representation of a mine, you can hover your mouse over a cell to see its ID show up in the tooltip.
- Each MapCell is one of the following types: start, exit, floor, wall, lava, gold, key (red, green, or blue), or locked door (red, green, or blue). There are *is*___ methods for each of these to indicate the type of cell. Each of these methods returns a boolean value. For example, isStart() will return true if the cell is a starting cell or false otherwise.
- For key cells and locked door cells, there are methods isKey() and isLock() to indicate the cell type. Note that these do not identify the colour of the key or lock. These should be used in conjunction with the methods isRed(), isGreen(), and isBlue() to determine the colour of the key or lock. Likewise, these colour methods do not identify whether the cell is a key or a lock but only whether or not they are of the specified colour. For example, consider a cell that contains a blue key. The methods isKey() and isBlue() would return true on this cell. The combination of those results is required to understand its identity.
- A MapCell can be marked as ***in-stack*** which indicates that the cell has been used as part of a current ongoing path. A MapCell can be marked as ***out-of-stack*** which indicates that it has been previously used in a path but is no longer part of the ongoing path. All MapCells begin with both statuses inStack and outStack equal to false. The methods markInStack() and markOutStack() should be used when writing your pathfinding algorithm to indicate the status of the cell. The boolean methods isMarkedInStack() and isMarkedOutStack() indicate that status. Additionally, the method isMarked() will indicate whether the cell has been marked in **either** way (in-stack OR out-of-stack). These methods will also be helpful when writing your pathfinding algorithm; they are named as they are because, as part of the algorithm to find the path to the exit, a stack of MapCells will be used.
- Each MapCell has two, three, or four neighbours (corner cells have two, edge cells have three, and internal cells have four). As explained earlier in this document, the indexing of neighbours is always such that 0 is the index for the neighbour to the top, 1 is the index for the neighbour to the right, 2 is the index for the neighbour to the bottom, and 3 is the index for the neighbour to the left. Use the getNeighbour(index) method to obtain a neighbouring MapCell object. Remember to check if the return value from this method is null. The cells that have two or three neighbours will obtain null for those neighbouring indices in which there is no MapCell.
- The method changeToFloor() is a method you will need to use in your path-finding algorithm to change the given cell to a floor cell. This is important when picking up a piece of gold, picking up a key, and unlocking a locked door. In these three cases, the cells will have to convert from their initial cell type into a floor cell type using this method.

Assignment 3

Classes to Implement

For this assignment, you must implement two (2) Java classes: **ArrayStack** and **MineEscape**. Follow the guidelines for each one below.

In these classes, you may implement more private (helper) methods if you want. However, you may not implement more public methods **except** public static void main(String[] args) for testing purposes (this is allowed and encouraged).

You may **not** add instance variables other than the ones specified in these instructions nor change the variable types or accessibility (i.e. making a variable public when it should be private). Penalties will be applied if you implement additional instance variables or change the variable types or modifiers from what is described here.

You may **not** import any Java libraries such as `java.util.Arrays`, `java.util.Stack`, or `java.util.ArrayList`.

ArrayStack.java

This class represents a Stack collection implemented using an array. It must implement the StackADT interface (which is provided to you). In this implementation, the top variable is an integer that tracks where the top of the stack is. The top variable must be -1 when the stack is empty and index the ***position of the last added element*** in the stack when the stack is not empty. (Note that this is different from how it was handled in class.)

The class must have exactly (no more and no less) the following private instance variables:

- private T[] array (this array holds the items in the stack)
- private int top (see above explanation of top for this implementation)

The class must have the following public methods:

- public ArrayStack (): constructor
  - Initialize the array with a capacity of 10
  - Initialize top to -1
- public void push (T element)
  - Expand the capacity of the array by adding 10 more spots if you are using greater than or equal to 0.75 (75%) of the array's capacity (see expandCapacity explanation below)
  - Add the element to the top of the stack and update the value of top
- public T pop () throws StackException
  - If the stack is empty, throw a StackException with the message "Stack is empty"
  - Then, shrink the capacity of the array by removing 10 spots if you are using less than or equal to 0.25 (25%) of the array's capacity AND the capacity is greater than or equal to 20 (see shrinkCapacity explanation below)
  - Finally, (after completing the shrink capacity process if it was needed) remove and return the element from the top of the stack, and update the value of top

- public T peek () throws StackException
    - If the stack is empty, throw a StackException with the message "Stack is empty"
    - Return the element from the top of the stack without removing it
- public boolean isEmpty ()
    - Return true if the stack is empty or false otherwise
- public int size ()
    - Return the number of elements in the stack
- public void clear ()
    - Clear out all elements from the stack and restore it to its original state (i.e., 10 empty spaces in the array and top = -1)
- public int getCapacity ()
    - Return the length (capacity) of the array
- public int getTop ()
    - Return the top index
- public String toString ()
    - If the stack is empty, return "Empty stack." (please ensure you include the period, otherwise your tests will fail).
    - Otherwise, build and return a string beginning with "Stack: " followed by all the items in the stack from the top (first) to the bottom (last). Entries should be separated by a comma, followed by a single space. The last element should be followed by a period (but no space).
- private void expandCapacity ()
    - Check the fraction of the array is being used (number of items in the array divided by the capacity). If that fraction is less than 0.75, do nothing and skip this step. (Hint: remember to cast the ints to doubles for the division to work properly)
    - If at least 0.75 (75%) is being used, expand the capacity by adding 10 additional spots in the array. Remember to follow the regular procedures for expanding an array, as demonstrated in class.
- private void shrinkCapacity ()
- Check the fraction of the array is being used (number of items in the array divided by the capacity). If that fraction is greater than 0.25 and/or the array's capacity is less than 20, do nothing and skip this step. (Hint: remember to cast the ints to doubles for the division to work properly)
- If at most 0.25 (25%) is being used and the capacity is at least 20, shrink the capacity by removing 10 spots from the array. Use the expandCapacity procedure as a guide for implementing the shrinkCapacity method.

MineEscape.java

This class is used to find the escape path out of the mines. A mine will be loaded in from a text file and this class must implement the algorithm to help us find our way out of the mine based on the layout of the mine and by following the set of rules and requirements for movement.

The class must have exactly (no more and no less) the following private instance variables:

- private Map map (the map of the current mine)
- private int numGold (the count of how many chunks of gold you are holding)
- private int[] numKeys (a count of how many red, green, and blue keys you are holding)

The class must have the following public methods:

- public MineEscape (String filename): constructor
  - Initialize the map variable using the given filename. To do this, create a Map object and pass it the filename variable as a parameter to its constructor.
  - Initialize numGold to 0 and numKeys as an array with 3 cells, all containing 0
  - Wrap the above lines of code into a try-catch statement to catch any Exceptions that may occur from the Map class. If an exception occurs, print the message from the exception.
- private MapCell findNextCell (MapCell cell)
  - Determine the next cell to walk onto from the current cell. To determine the next cell, you must follow the list of rules explained above in the **Movement Rules and Restrictions** section.
  - Return the MapCell object representing the next cell to walk onto from the current cell if one exists; otherwise return null
- public String findEscapePath ()
  - Determine the path from the starting point to the exit cell, if one exists, using the findNextCell() helper method and using the algorithm from the pseudocode given below.
  - To get the start cell, use the getStart() method from the map instance variable. As you walk through each mine, build a path string that starts with "Path: " and then contains each cell's ID (i.e. use getID() on the MapCell objects) that you walk on with a space after each. At the end of the algorithm, append the number of gold chunks being held and a "G" to finish the path string. If a possible path exists, return the path string; otherwise return "No solution found".

Escape Path Algorithm Pseudocode

```
initialize the ArrayStack S to store MapCell objects
push the starting cell onto S
set a boolean variable running to be true
mark the starting cell as in-stack
while S is not empty and running is true
    curr = peek at S
    if curr is the exit cell, set running = false and end the loop immediately
    if curr is a key cell, determine its colour and update numKeys accordingly* (see note below)
    if curr is a gold cell, update numGold accordingly* (see note below)
    if curr is adjacent to lava, reset numGold to 0
    next = findNextCell(curr)
    if next = null, set curr = pop off stack and mark curr as out-of-stack
    else
        update path string by adding next
        push next onto S
        mark next as in-stack
        if next is a locked door cell
            determine colour of locked door
            unlock the door** and update numKeys accordingly* (see note below)
if running is false
    return path (including gold count)
else
    return "No solution found"
```

* When picking up a piece of gold, a key, or unlocking a door, it is important that you call the method changeToFloor() on the cell that was containing the gold, key, or locked door. This method changes it to a floor cell to visually indicate that the item has been removed and it's functionally required to ensure the cell is no longer treated as its original type.

** this is based on the assumption that we have the key of the correct colour – this should be checked in the findNextCell() method so it doesn't have to be re-checked here.

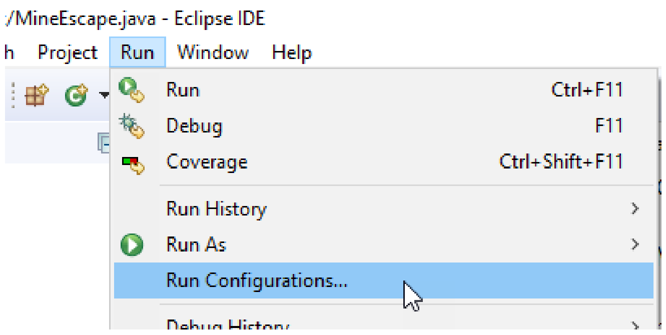
Testing and Debugging

The TestPath file runs all 10 provided mine maps sequentially. It may help for you to first run MineEscape so that you can run it with one map at a time and watch the animated movements throughout each mine. This will be helpful for testing and debugging your path-finding algorithm. To do this, you need to include the main method that is provided below into the MineEscape class AND you need to set a command line argument in the Run Configurations (in Eclipse) to indicate which mine file you want to load. The steps to do this are on the next page of this document.

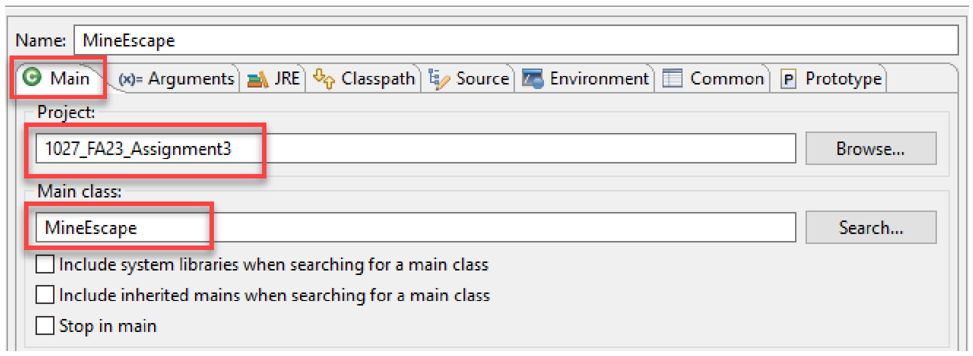
```
    public static void main(String[] args) {
          if (args.length != 1) {
                System.out.print("Map file not given in the arguments.");
          } else {
                MineEscape search = new MineEscape(args[0]);
                String result = search.findEscapePath();
                System.out.println(result);
          }
    }
```

The following steps show you how to set a Command Line Argument in Eclipse. If you are using a different IDE, you may have to look up how to set the command line arguments in that IDE.

Click Run > Run Configurations…

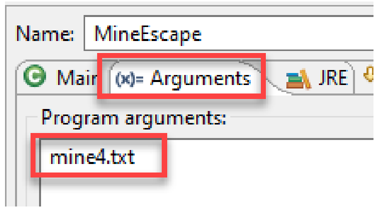


In the "Main" tab, select the correct Project and the "Main class" you wish to run (MineEscape in this case).



Name: MineEscape
Project: 1027_FA23_Assignment3
Main class: MineEscape
☐ Include system libraries when searching for a main class
☐ Include inherited mains when searching for a main class
☐ Stop in main

In the "Arguments" tab, type in the argument you want to be sent in to the main method (in this case, a mine text file name such as mine4.txt)



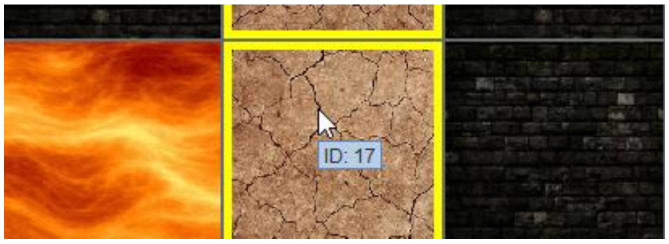
Name: MineEscape
Program arguments: mine4.txt

Press Apply and then Run. It will start running the class with the given argument sent in to your main method.

Note that every time you want to change the mine file being loaded into MineEscape, you will have to go back through these steps and change the filename in the Run Configurations.

Additional testing and debugging strategies:

- Try creating your own mine map and run them from MineEscape as well. Observe the format of the provided mine maps to help you understand how to create your own. Make sure your algorithm works in a variety of different scenarios.
- While debugging your code, use print lines to display different variable values to help understand why code may be behaving the way it is. Print lines also help to see the flow of execution of the program. Adding checkpoint print lines can help you see which methods are being called.
- Each cell in a mine is given a unique int ID. It may help to print out the ID of a cell from your algorithm using the getID() method on any MapCell object. You can also see the cells' IDs in a tooltip when you hover your cursor over a cell (see example below).



Marking Notes

Functional Specifications

- Does the program behave according to specifications?
- Does it produce the correct output and pass all tests?
- Are the classes implemented properly?
- Does the code run properly on Gradescope (even if it runs on Eclipse, it is **up to you** to ensure it works on Gradescope to get the test marks)
- Does the program produce compilation or run-time errors on Gradescope?
- Does the program fail to follow the instructions (i.e. changing variable types, etc.)

Non-Functional Specifications

- Are there comments throughout the code (Javadocs or other comments)?
- Are the variables and methods given appropriate, meaningful names?
- Is the code clean and readable with proper indenting and white-space?
- Is the code consistent regarding formatting and naming conventions?
- Submission errors (i.e. missing files, too many files, etc.) will receive a penalty.
- Including a "package" line at the top of a file will receive a penalty.

Assignment 3

CS 1027
Computer Science Fundamentals II

Remember **you must do** all the work on your own. **Do not copy** or even look at the work of another student. All submitted code will be run through similarity-detection software.

Submission (due Tuesday, November 14 at 11:55 pm)

Assignments must be submitted to Gradescope, not on OWL. If you are new to this platform, see these instructions on submitting on Gradescope.

Rules

- Please only submit the files specified below.
- Do not attach other files even if they were part of the assignment.
- Do not upload the .class files! Penalties will be applied for this.
- Submit the assignment on time. Late submissions will receive a penalty of 10% per day.
- Forgetting to submit is not a valid excuse for submitting late.
- Submissions must be done through Gradescope. **If your code runs on Eclipse but not on Gradescope, you will NOT get the marks! Make sure it works on Gradescope to get these marks.**
- You are expected to perform additional testing (create your own tester class to do this) to ensure that your code works for a variety of cases. We are providing you with some tests but we may use additional tests that you haven't seen for marking.
- Assignment files are NOT to be emailed to the instructor(s) or TA(s). They will not be marked if sent by email.
- You may re-submit code as many times as you wish, however, re-submissions after the assignment deadline will receive a late penalty.

Files to submit

- ArrayStack.java
- MineEscape.java

Grading Criteria

Total Marks: 20

15 marks	Autograder Tests (some tests are hidden from you)
5 marks	Non-functional Specifications (code readability, comments, correct variables and functions, etc.)