

# Assignment 3

Due date: March 19 at 11:55 pm

## 1. Learning Outcomes

In this assignment, you will get practice with:

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

## 2. Introduction

Middlesex College (MC) is the most beautiful building on campus and it is the home to the Computer Science department. You are somewhere at Western and need to find your way to MC but you have to watch out for all the angry geese scattered around campus! You also should try to avoid the snow piles that can get your feet cold and wet. While on your way, if you happen to see any lost books, please pick them up and bring them to MC with you – someone might give you a finder's fee for those books they lost.

In this assignment, you have to first implement a Stack and once you have that working (make sure it works BEFORE moving on to the second part), you must write an algorithm to navigate campus using the Stack you just created to keep track of movements and allow you to back-track when you get too close to a gaggle of vicious geese. There are several rules you have to follow to ensure that the path you take is the right one (not necessarily the shortest one).





There are a variety of campus maps, each stored in an individual text file that is being read in and loaded in as a map you can visually see if you wish. Each map is divided into a series of hexagonal cells interconnected such that each cell is connected to its up-to-6 neighbours (not all cells will actually have 6 neighbours if they are around the boundary of the map) and each cell is of one specific type. All the cell types are described and shown in Figure 1. Each cell also has an ID which is shown near the top-middle of the cell in the visual program.

Your algorithm must allow you to walk from one cell to a neighbouring cell one by one until you reach MC. In some cases, there might not be a valid path to MC in which case you have to backtrack all the way to the starting point. As you walk, you must adhere to the rules explained below so that you are choosing the best cell to walk onto based on the cell type of your current cell and of all its neighbouring cells.

To avoid walking in circles, you can never walk on the same cell more than once **except when backtracking in which case you will have to "undo" steps which will bring you back to cells you previously walked on.**

# Assignment 3

## Cell Types

<b>Grass Cell</b> You can walk freely on grass cells.	
<b>Start Cell</b> You begin on the start cell and it behaves like a regular grass cell	
<b>End Cell</b> Your goal is to reach the end cell (Middlesex College).	
<b>Goose Cell</b> You cannot walk on a goose cell – avoid being attacked. You should also be careful when walking adjacent to goose cells!	

# Assignment 3



<p><b>Snow Cell</b></p> <p>Snow cells are a last resort – you should avoid them if you can, but you can take them if absolutely necessary.</p>	
<p><b>Book Cell</b></p> <p>A book cell behaves like a normal grass cell but you pick up the book when you go through it.</p>	

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.

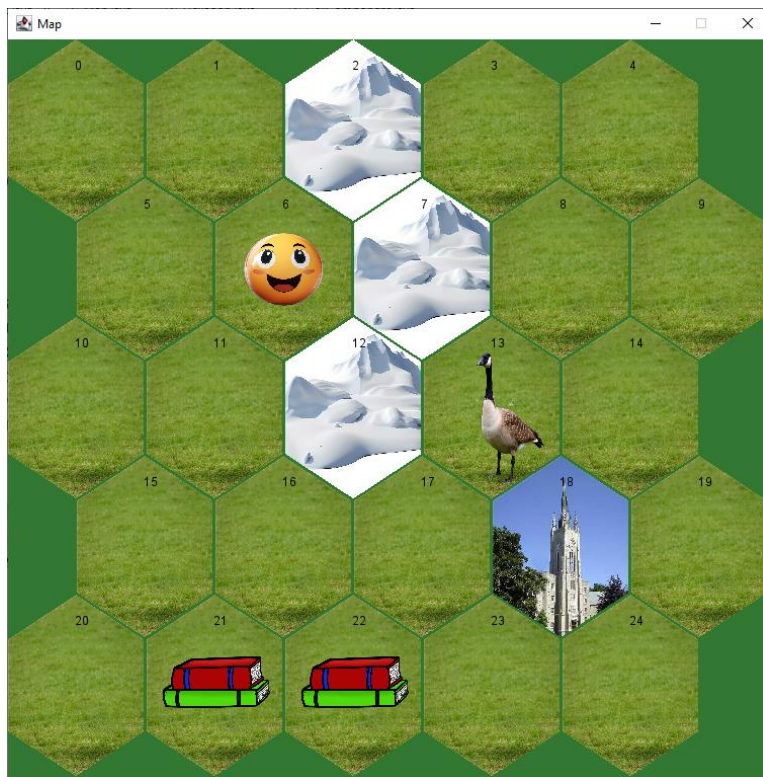


Figure 2. A sample map including each of the different cell types. The start cell has the ID #6 and the end cell has ID #18. There are book cells at IDs #21 and 22 and snow cells at IDs #2, 7, and 12. All other cells in this map are grass cells.

# Assignment 3

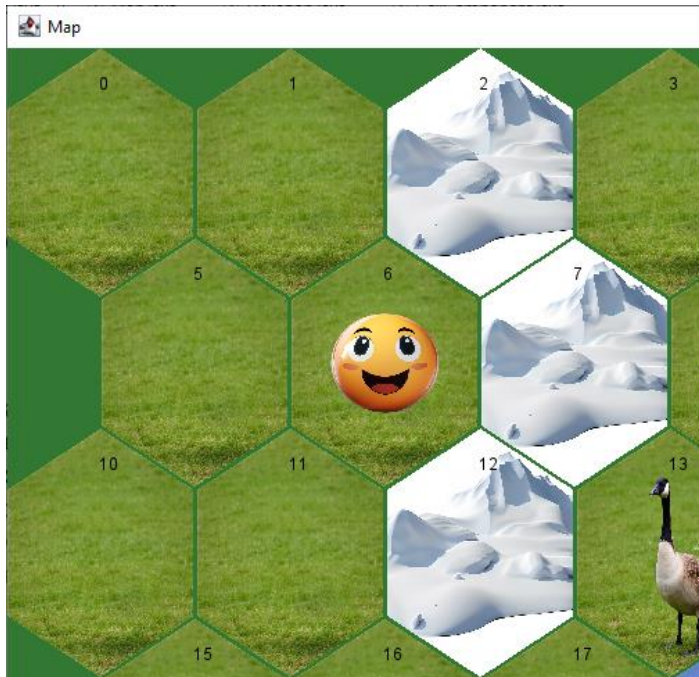


Figure 3. A closer look at the start cell and its neighbour cells from the map shown in Figure 2.



Figure 4. The start cell and its neighbour cells from the map shown in Figure 2, but now the neighbour cells are shown with their neighbour indices from the start cell. Neighbour indices always begin at 0 in the cell north-east of a given cell and the indices increase in clockwise order around the cell.



# Assignment 3

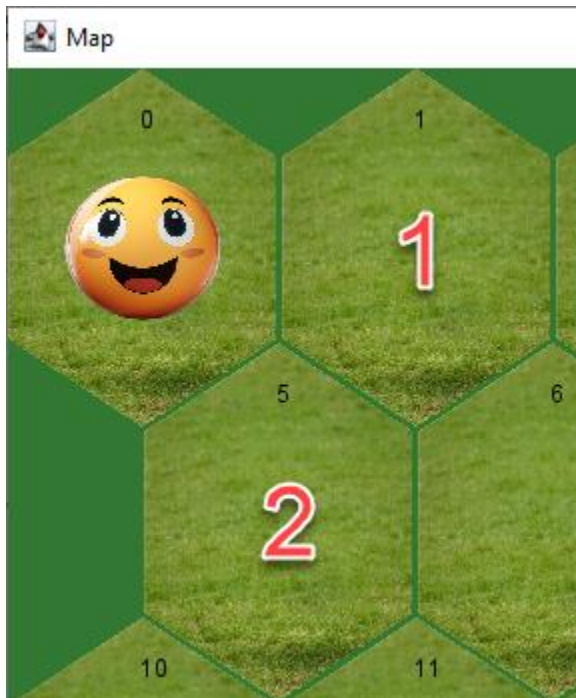


Figure 5. A different map in which the start cell is along the map's boundary and thus does not have 6 neighbours. In this case, there are only 2 neighbours from this start cell. Notice that the indices of these neighbours are 1 and 2. Neighbour index 0 would be north-east of the start cell but in this case it is null since it's out of the map. Neighbour indices 3-5 are also null since they are out of the map.

## Movement Rules and Restrictions

Your task is to use an algorithm to find your way from the start cell to the end cell (Middlesex College) in each map. The algorithm must use a stack (specifically the `ArrayStack` class that you are expected to implement yourself) to keep track of unexplored locations in the map. Part of this main algorithm is a smaller algorithm that defines how to move from one cell to the next cell in a map, which is described 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 Middlesex College exit in the same map. Additionally, there are rules that refer to the markings on cells: as you walk through a map, 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 (but remember that when backtracking, you are retracting movements and thus walk back on some of the cells you took to arrive at that point so you may be on the same cell multiple times when backtracking).

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:

# Assignment 3

## CS 1027 Computer Science Fundamentals II

- the next cell is not a goose cell
- the next cell does not have a goose-neighbour-count (see below) of 3 or more (**EXCEPTION:** when going to the exit cell, the goose-neighbour-count can be ignored since you can run into the building without the geese attacking you, but in **ALL** other cases this is a strict rule)
- 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 end cell, go to the end cell
  2. if curr is adjacent to one or more cells that contain a book, go to the book cell neighbour with the smallest index (0-5)
  3. if curr is adjacent to one or more grass cells, go to the grass cell neighbour with the smallest index that has the lowest goose-neighbour-count (see below)
  4. if curr is adjacent to one or more snow cells, go to the snow cell with the smallest index
  5. if none of these conditions are met, return null to indicate that you cannot proceed and must backtrack

\* The "**goose-neighbour-count**" is the number of goose cells adjacent to a given cell.

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

### 3. Provided files

You are given several Java files and other types of files for this assignment:

1. CellComponent.java
2. Hexagon.java (\* see more information below \*)
3. HexLayout.java
4. Map.java
5. StackADT.java (\* this is the interface you must use when implementing ArrayStack \*)
6. Several custom exception files:
  - CollectionException.java
  - IllegalArgumentException.java
  - InvalidMapException.java
  - InvalidNeighbourIndexException.java
7. Several text files (map1.txt, map2.txt, map3.txt)
8. Several image files (\*.jpg)

Save all the Java files into the src folder of your project folder for this assignment. Save all text files and image files into the project's root folder (not inside bin or src). Most IDEs will expect

them outside of those sub-folders, although you may have to check with the configuration of your IDE.

Do **not** edit any of the provided files.

## More Information About Hexagon

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

- Each Hexagon object has a unique int ID. This can be obtained using `getID()`. This is the ID you see near the top-middle of the Hexagons in the visual program. You can also hover your mouse over a cell to see its ID show up in the tooltip.
- Each Hexagon is one of the following types: start, end, grass, snow, book, or goose. 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.
- A Hexagon can be marked as ***in-stack*** which indicates that the cell has been used as part of a current ongoing path. A Hexagon 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 Hexagons 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 Hexagons will be used.
- When a Hexagon is marked as ***in-stack***, it is shown with a green border. When it is marked as ***out-of-stack***, it is shown with a red border. When watching the pathfinding algorithm in the visual program, you can see the Hexagon border colours to determine the path being taken.
- Each Hexagon has up to 6 neighbours (boundary cells have less than 6 neighbours). As explained earlier in this document, the indexing of neighbours is always such that 0 is the index for the neighbour to the north-east, 1 is the index for the neighbour to the east, and so on in clockwise order. Use the `getNeighbour(index)` method to obtain a neighbouring Hexagon object. Remember to check if the return value from this method is null. The cells that have less than 6 neighbours will obtain null for those neighbouring indices in which there is no Hexagon.
- `Hexagon.TIME_DELAY` is a public static variable that you can change to adjust how the speed of the pathfinding algorithm animation. This is the delay time in milliseconds between one movement and the next. When debugging, it may help to increase this value (maybe to 1000 or 2000, e.g.) to better track each step being taken.

## 4. Classes to Implement

For this assignment, you must implement two Java classes: **ArrayStack** and **CampusWalk**. 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.ArrayList`, or `java.util.Stack`. Doing so will result in strict penalties.

### 1) 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 elements must be added to the right-most available cell in the array (i.e. the first item must be at index  $n-1$  where  $n$  is the length of the array) and the top variable is an integer that represents the index of the next available cell from the right toward the left. The top variable must be  $n-1$  when the stack is empty and  $-1$  when the array is full (i.e. all the cells are storing elements). Note that this is different than the implementation taught in class.

This class must have the following private instance variables (no additional global variables are allowed):

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

This class must contain the following methods (no additional methods, other than private helper methods, are allowed):

- Constructors
  - `public ArrayStack()`
  - `public ArrayStack(int initCapacity)`
  - In the first constructor, initialize the array with a capacity of 10; in the second constructor, initialize the array with a capacity of *initCapacity*. In both constructors, initialize top as explained above.
- `public void push (T element)`



- Check if the array is full and, if so, expand its capacity (see `expandCapacity` explanation below).
  - Add the *element* to the top of the stack (rightmost available cell in the array) and update the value of `top`.
- `public T pop ()` throws `CollectionException`
  - If the stack is empty, throw a `CollectionException` with the message "Stack is empty".
  - Remove and return the element from the top of the stack, and update the value of `top`.
- `public T peek ()` throws `CollectionException`
  - If the stack is empty, throw a `CollectionException` 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 int getCapacity ()`
  - Return the length (capacity) of `array`.
- `public int getTop ()`
  - Return the `top` index.
- `public String toString ()`
  - If the stack is empty, return "Empty stack."
  - Otherwise, build and return a string containing all the items in the stack starting from the top (first) to the bottom (last). Entries should be separated by a comma and a single space.
- `private void expandCapacity ()`
  - This is a helper method that should only be called when the array is full and another item needs to be pushed on. If the current array capacity is 15 or less, then expand it by doubling its current capacity. Otherwise, expand it by adding 10 additional spaces in the array.
  - Remember to follow the regular procedures for expanding an array, as demonstrated in class. Also ensure that the items are stored in the rightmost cells of the array as specified for this implementation and that the order is preserved (i.e. do not change the order of items when expanding the capacity).

## 2) CampusWalk.java

This class is used to find the path from the starting point to Middlesex College. A map will be loaded in from a text file and this class must implement the algorithm to help us find our way through the map based on the layout of the map and by following the set of rules and requirements for movement.

# Assignment 3

## CS 1027 Computer Science Fundamentals II

This class must have the following private instance variable (no additional global variables are allowed):

- private Map `map`

This class must contain the following methods (no additional methods, other than private helper methods, are allowed):

- Constructor
  - public CampusWalk (String `filename`, boolean `showMap`)
  - Initialize the Map object with the given `filename`
  - Surround this line with a try-catch structure and print out a simple message such as "Error occurred" if any type of exception is caught in here.
  - If `showMap` is true, call `map.showGUI()`; to open the visual program. Otherwise, call `map.hideGUI()`; (or do nothing as the map should be hidden by default).
- public int neighbourGooseCount (Hexagon `cell`)
  - Count and return the number of goose cells surrounding the given `cell`. Hint: remember that not all cells have 6 neighbours so check if a neighbour is not null before checking its type.
- public Hexagon findBest (Hexagon `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 Hexagon object representing the next cell to walk onto from the current `cell` if one exists; otherwise return null.
- public String findPath ()
  - Determine the path from the starting point to the end cell (Middlesex College), if one exists, using the findBest() 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 map, build a path string that contains each cell's ID (i.e. use getID() on the Hexagon objects) that you walk on with a space after each. If a possible path exists, return the path string; otherwise return "No path found".
- public void exit ()
  - Call `map.exit()`; (this is useful to force the map to be closed when a test case is completed)

### Escape Path Algorithm Pseudocode

```
initialize Stack S
push the starting cell onto S
set a boolean variable running to be true
mark the starting cell as in-stack
```

# Assignment 3

## CS 1027 Computer Science Fundamentals II

```
while S is not empty and running is true
    curr = peek at S
    update the path string with curr's ID
    if curr is the exit cell, set running = false and end the loop
immediately

    next = findBest(curr)
    if next = null
        pop off stack
        mark curr as out-of-stack
    else
        push next onto S
        mark next as in-stack
if running is false
    return path string
else
    return "No path found"
```

### Testing and Debugging

Run CampusWalk with a given map file and showMap = true so that you can watch the path algorithm go one step at a time. 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 CampusWalk class.

```
public static void main(String[] args) {
    Hexagon.TIME_DELAY = 500; // Change speed of animation.
    String file = "map1.txt"; // Change when trying other maps.
    CampusWalk walk = new CampusWalk(file, true);
    String result = walk.findPath();
    System.out.println(result);
}
```

Create your own maps using the format in the provided files but with different layouts. It's important to check that your algorithm works in a variety of different cases.

## 5. 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.

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. The typical penalty for cases of academic dishonesty on an assignment is a mark of 0 on the assignment and a report sent to the Dean's Office. The following are all forms of academic dishonesty:

- Sharing/sending/showing your code to one or more peers
- Posting your code online (including repositories such as GitHub)
- Looking at another student's code
- Leaving your computer unattended in a public location (such as a library or classroom)
- Using ChatGPT or other generative AI platforms to generate any amount of code or comments for you
- Paying someone to write code for you
- Using code you find online or in a textbook

### Submission (due Wednesday, March 19 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.

# Assignment 3

## CS 1027 Computer Science Fundamentals II

### 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 result in late coupons being applied while they last. Once they are all used, late submissions will receive a penalty of 10% per late day. Submissions will not be accepted more than 3 days after the due date regardless of the number of late coupons you have.
- 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 your own tests to ensure the correctness of your program.
- 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 be considered late and handled as such.
- Every file you create for this assignment must include a header including your name, student number, email address, the course code, assignment number/name, and the date on which you first created the file (you don't need to update it when you modify the file). i.e. (the formatting doesn't have to be the same as this example)

```
/*  
CS 1027B - Assignment 1  
Name: Tom Cruise  
Student Number: 123456789  
Email: topgun@uwo.ca  
Created: July 1, 2025  
*/
```

### Files to submit

- ArrayStack.java
- CampusWalk.java

## 6. Grading Criteria

Total Marks: [20]

### Functional Specifications:

[1] Compilation (your code must compile on Gradescope)



# Assignment 3

## CS 1027 Computer Science Fundamentals II

[15] Passing Tests (they must run and pass on the Gradescope autograder – we will not be manually grading or modifying these grades)

### **Non-Functional Specifications:**

- [1] Header information at the top of all classes you create
- [1] Code formatting (readability, indentation, descriptive variable names)
- [1] Code comments
- [1] Consistent naming convention (e.g. camelCase or snake\_case).