CS1027

ASSIGNMENT 3

Computer Science Fundamentals II

This assignment is due on November 22nd, 2020 by 11:55pm.
See the bottom of this document for submission details.

Learning Outcomes

To gain experience with

- Solving problems with the queue data type
- Using inheritance to reuse & modify existing code
- The design of algorithms in pseudocode and their implementations in Java
- Handling exceptions

Introduction

Halloween isn't quite over for our intrepid adventurer from Assignment 2. After having fallen through a magical portal during his trick-or-treating, Bryan the Bold has found himself lost and without his candy! Worse yet, this strange new realm has actual monsters!

After some investigating, Bryan found a helpful Sorceress that was willing to send him back to his home, but she requires payment in candies for the spell she's going to cast. Using your path-finding skills from last time, Bryan needs your help to figure out which parts of this strange new world are worth the risk to investigate, and which ones are not. He has some maps of locations to explore for candy, but wants to plan out whether they are safe to explore or not.

Stranger yet, when Bryan gets to the candy piles, he briefly unlocks some superpowers in this strange new realm! When he picks them up, he's able to throw lightning bolts at the approaching monsters! Between this and some smart planning, he's hoping to collect enough candy to pay the Sorceress for her services.

Can you help Bryan collect enough candies to return home?

Reminder: Valid Paths

When looking for a path the program must satisfy the following conditions:

- The path can go from Bryan, a cross path cell, or a candy cell to the following neighbouring cells:
    - A candy cell,
    - A cross path cell,
    - The north cell or the south cell, if such a cell is a vertical path, or

  - The east cell or the west cell, if such a cell is a horizontal path.

- The path can go from a vertical path cell to the following neighbouring cells:
  - The north cell or the south cell, if such a cell is either Bryan, a cross path cell, a candy cell, or a vertical path cell.
- The path can go from a horizontal path cell to the following neighbouring cells:
  - The east cell or the west cell, if such a cell is either Bryan, a cross path cell, a candy cell, or a horizontal path cell.

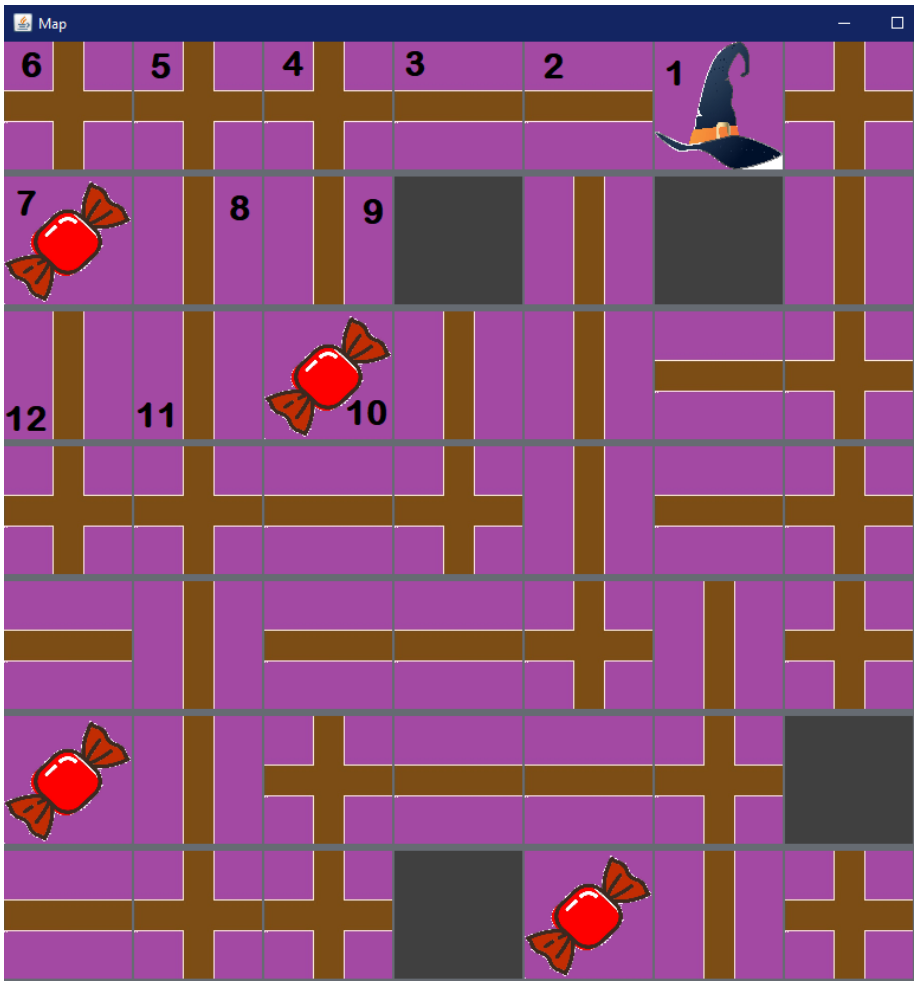


Figure 1. A sample of one of Bryan's neighbourhood maps represented as a grid of cells

PC vs NPC: 'PC' means 'Player Character', and refers to the character usually controlled in a video game or board game. 'NPC' refers to Non-Player character and is controlled by the game. Here, Bryan's token is the 'PC' and the Zombies are the NPCs.

NEW NPC Pathing:

If while looking for a path the program finds that from the current cell there are several choices as to which adjacent cell to use to continue the path, your program must select the next cell for the path in the following manner:

- The NPC prefers a PC cell over all other cells.
- The NPC then prefers a candy cell over the other cells.
- If there is no candy cell adjacent to the current cell, then the NPC must prefer the cross path cell over the other cells; and
- If there is no cross-path cell, then the NPC chooses the smallest indexed cell that satisfies the conditions described above.

- NPCs do not mark their path and may end up in a loop – this is okay.

Provided files

The following is a list of files provided to you for this assignment. You do not need alter these files in any way and should assume we will mark them with unmodified ones. Studying these will help improve your understanding of Java. Some of these will be familiar from Assignment 2, and we will only go over the new methods you will need to implement your classes:

- Map.java – This class represents the map of the neighbourhood, including Bryan's starting location and the candy houses. New methods you may see include but are not limited to:

    - *public MapCell[] getZombieCells(int numZombies)*
    - This method returns the cells that Zombies will spawn in and is used in StartSearch.java to put the zombies in their starting position at the bottom of the map.

- MapCell.java – This class represents the cells of the map. Objects of this class are created inside the class *Map* when its constructor reads the map file. The new methods that you might use from this class are:

    - MapCell.CellType getType() returns the type of the current cell, which is useful for figuring out if a PC or NPC token has collided with each other, or for getting the cell that a PC or NPC token is obscuring.
    - void setType(MapCell.CellType) takes in one of the pre-set types (HERO, SUPERHERO, ZOMBIE, GHOST) and sets the MapCell to be of that type. Used

when a PC or NPC token moves onto a new cell and the display needs to be updated, and the one left behind needs to be returned to its original cell type.
    - You may also wish to check out CellType in this class.
    - **Forgetting to set types before and after a token moves is one of the most common causes of bugs!**

- StartSearch.java
    - This file has been modified from the A2 solution to handle multiple object types.
    - You should not modify this file. Code your required .java files to match and allow this class to compile.

- PlayerObject.java
    - This class moves the solution from A2 for finding the best cell and uses it to move the Player object around the map like before.
    - Studying this class and modifying it to suit NPCObject.java is strongly recommended.
    - You are asked to implement specific parts of this class – see below.

- Other classes provided:
    - *ArrayStack.java, ArrayStackADT.java, QueueADT.java, CellColors.java, CellComponent.java, InvalidNeighbourIndexException.java, CellLayout.java, InvalidMapException.java, IllegalArgumentException.java, EmptyCollectionException.java, EmptyStackException.java, TestQueue.java, TestSearch.java (Updated)*

Classes to implement

A description of the classes that you are required to implement for full marks in this assignment are given below. You may implement more classes if you want; you must submit the code for these with your assignment.

In all these classes, you may implement more private (helper) methods as desired, but you may **not** implement more public methods. You may **not** add static methods or instance variables. Penalties will be applied if you implement these.

For this assignment, you may not use Java's *Queue* class, although reading and understanding the code there may help you better understand queues. You also may not use any other pre-made Java collections libraries. The data structure required for this assignment is a circular queue, described below:

CircularArrayQueue.java

This class represents a Queue implementation using a circular array as the underlying data structure. This class must implement the QueueADT and work with the generic type (T). Note: the front index must start at DEFAULT_CAPACITY-1 and rear must start at DEFAULT_CAPACITY-2

This class must have the following *private* variables:

- front (int)
- rear (int)
- count (int)
- queue (T array)
- DEFAULT_CAPACITY (final int) with a value of 20

The class must have the following *public* methods:

- CircularArrayQueue (constructor) – no parameters required in this first constructor. Initialize the front to DEFAULT_CAPACITY-1, rear to DEFAULT_CAPACITY-2, count to 0, and the queue array using the final int variable as the array's capacity.
- CircularArrayQueue (second constructor) – same as the first constructor described above, except that this one takes in an int parameter for the initial capacity rather than using the default capacity. Front and rear are set to initialCapacity-1 and initialCapacity-2 respectively.
- enqueue – takes in an element of the generic type and adds that element to the rear of the queue. If the queue is full before adding this item, then call expandCapacity.
- dequeue – throws an EmptyCollectionException if the queue is empty; otherwise remove and return the item at the front of the stack.
- first – throws an EmptyCollectionException if the queue is empty; otherwise return the item at the front of the queue without removing it.
- isEmpty – returns true if the queue is empty, and false otherwise.
- size – returns the number of items on the queue.
- getFront – returns the front index value (NOTE: this is not part of the QueueADT but is still required for this assignment).
- getRear – returns the rear index value (NOTE: this is not part of the QueueADT but is still required for this assignment).
- getLength – returns the current length (capacity) of the array (NOTE: this is not part of the QueueADT but is still required for this assignment).
- toString – returns the string containing "QUEUE: " followed by each of the queue's items in order from front to rear with ", " between each of the items and a period at the end of the last item. If the queue is empty then print "The queue is empty" instead.
- expandCapacity (private) – create a new array that has 20 more slots than the current array has, and transfer the contents into this new array and then point the queue instance variable to this new array by resetting the front and rear appropriately. You may

set the front to 1 and the rear to count when expanding the array instead of the DEFAULT_CAPACITY based front and rear.

NPCObject.java

This class will be used to represent one or more enemy tokens on the map. It is similar to PlayerObject.java, but used to allow the enemy tokens to have unique behaviours from it.

This class must have the following *private* variables:

- private CircularArrayQueue<MapCell> npcMoveQueue;
- private MapCell cell;
- private MapCell.CellType onTopOfCellType
- private boolean heroCollision;

This class must have the following methods (some private, some public):

- public NPCObject (MapCell startCell)
  - Creates a new NPC object, sets cell to startCell, sets the cellType to 'ZOMBIE', preserves the original cell type in onTopOfCellType, and initializes the CircularArrayQueue.
- Private MapCell singleMove(MapCell cell)
  - This is a helper method used by the following method. Given a single cell, it finds the next cell that is the best path for the NPC to take. Please see the section on 'NPC Pathing' for specifics on what must be changed for the NPC class pathing.
  - If the NPC queues up a cell of type HERO, set heroCollision to true.
- Public void queueMovePlan()
  - Zombies are slow to react, and this method enforces that by forcing them to plan their moves 3 steps in advance.
  - When the npcMoveQueue is empty, this method should queue up the 3 next best moves given the NPC's current spot on the map using singleMove(MapCell cell) above.
  - When the npcMoveQueue is not empty, this method should not adjust or interact with the queue in any way.
- Public void move()
  - This represents the zombie moving. You'll find that the zombie can find a pair of cells that it moves back and forth between – this is okay, they are zombies and can often get stuck in patrols.
  - This method dequeues the moves from queueMovePlan (one at a time per iteration of the loop in StartSearch.java)
  - Make sure to update the cell types once the NPC has moved from a square (back to its original type) to the new one (temporarily updated to ZOMBIE)

- Public Boolean checkHeroCollision(), publicMapCell getCell() & public MapCell.CellType getCellType() -- These are all getters for the variables in the class.

AdvancedPlayerObject.java

This class must inherit from Player object.

This class must have the following *private* variables:

- Private Boolean collision
- *Remember that as an inherited class it can use all of the parent class variables that are available to it!*

This class must have the following methods:

- Public AdvancedPlayerObject(MapCell startCell)
  - Follows the usual rules for inherited constructors
  - Suggested (not required, there are other ways to solve) to set onTopOfCellType = MapCell.CellType.SUPERHERO
  - If you don't have an issue getting the 'Superhero' icon to display this may not be required for your solution.
- Public MapCell getMove(MapCell newCell)
  - The advanced player object functions a bit differently than the normal one.
  - Firstly, it doesn't move, it only attacks.
  - This method will check all neighbours to see if they are NPC objects and smite the nearest one, using the index order when there are multiple options
  - If there are none in the neighbours, all of the neighbours **of** the neighbours are also checked.
- Public void smiteZombie(MapCell zombieCell, NPCObject[] zombieArray)
  - This method is used to imitate 'attacking' one of the NPCs once the Hero token has moved onto a candy tile.
  - Given a tile that has a zombie, find which entry in zombieArray corresponds to it, and then delete that zombie from the array. The model solution turns the entry to null, but there may be other ways to do this if you wish to investigate.
  - Print out an indication that an NPC was smote, and the identifier of the cell where it happened.

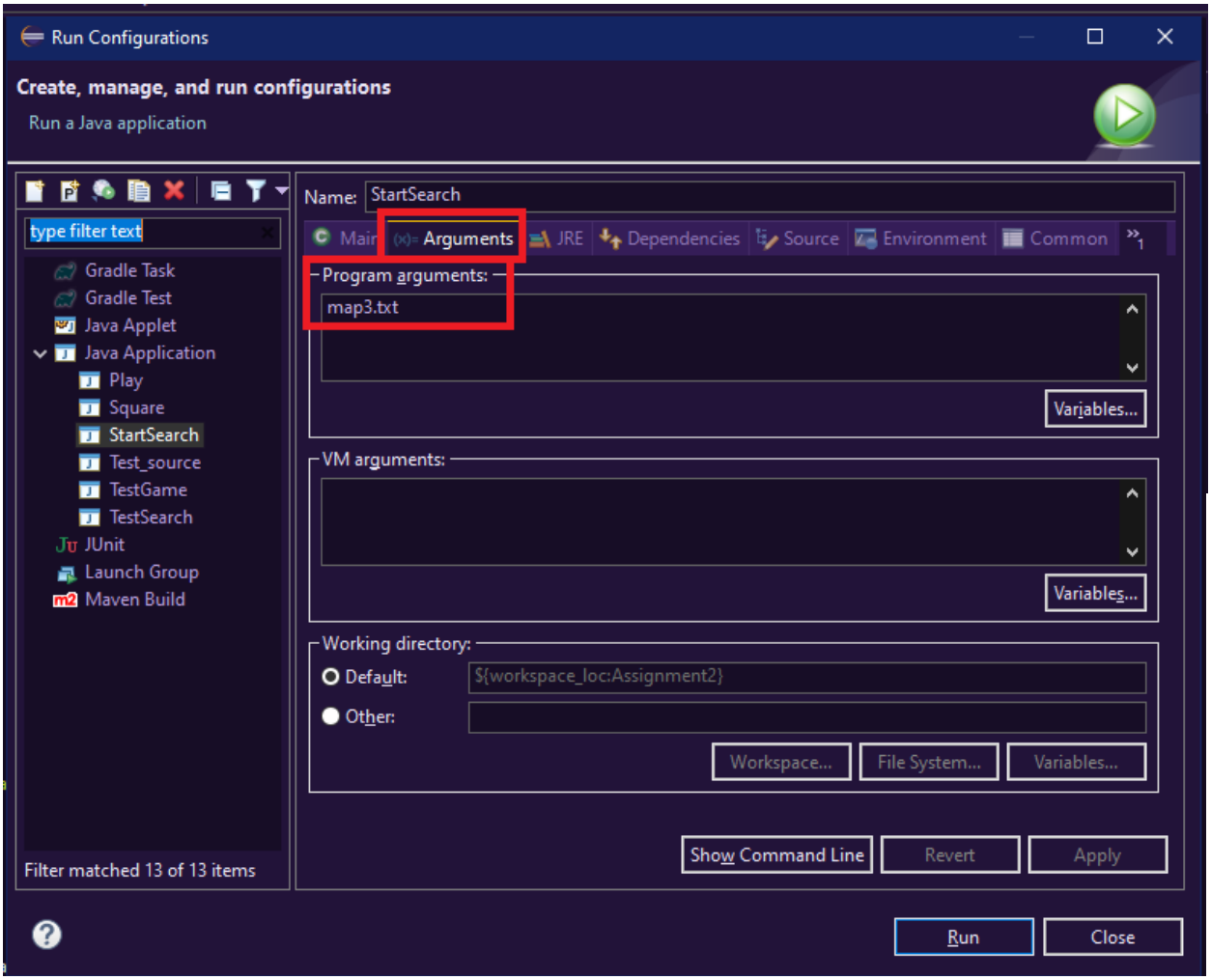
Command Line Arguments

Your program must read the name of the input map file from the command line.
You can run the program with the following command:

java StartSearch nameOfMapFile maxPathLength numberOfZombies

Where nameOfMapFile is the name of the file containing the desert map, and maxPathLength is the longest path that Bryan can take to find candies, and numberOfZombies is how many Zombies to spawn.

You can access run configurations as follows:



In the text box for 'Program arguments', you can type your arguments in. Ex: a3map1.txt 20 2, for running a3map.txt with a maximum path length of 20 and spawning 2 zombies. Note that the program sets a maximum amount of zombies based on the map size.

Image Files and Sample Input Files Provided

- You are given several image files that are used by the provided Java code to display the
- different kinds of map cells on the screen.

- You are also given several input map files that you can use to test your program.
- In Eclipse put all these files inside your project file in the same folder where the src folder is.
- Do not put them **inside** the src folder as Eclipse will not find them there.
- If your program does not display the map correctly on your monitor, you might need to move these files to another folder, depending on how your installation of Eclipse has been configured.

Marking notes

Marking categories

- Functional specifications
  - Does the program behave according to specifications?
  - Does it produce the correct output and pass all tests?
  - Are the class implemented properly?
  - Are you using appropriate data structures?
- Non-functional specifications
  - Are there Javadocs comments and other comments throughout the code?
  - 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?
- Penalties
  - Lateness: 10% per day
  - Submission error (i.e. missing files, too many files, ZIP, etc.): 5%
  - "package" line at the top of a file: 5%

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 cheating-detection software.

Submission (due November 22nd, 2020 at 11:55pm ET)

Rules

- Please only submit the files specified below. Do not attach other files even if they were part of the assignment.
- 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

- Submitting the files in an incorrect submission page or website will receive a penalty.
- 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 if your previous submission was not complete or correct, however, re-submissions after the regular assignment deadline will receive a penalty.

Files to submit

- CircularArrayQueue.java
- NPCObject.java
- AdvancedPlayerObject.java
- Any custom classes used.

Grading Criteria

- Total Marks: [20]
- Functional Specifications:
  - [3] CircularArrayQueue.java
  - [3] NPCObject.java
  - [2] AdvancedPlayerObject.java
  - [10] Passing Tests
- Non-Functional Specifications:
  - [0.5] Meaningful variable names, private instance variables
  - [0.5] Code readability and indentation
  - [1] Code comments