

# Assignment 3

**Due date: March 20 at 11:55 pm**

## Learning Outcomes

In this assignment, you will get practice with:

- Implementing an ADT
- Using stacks and priority queues to solve a problem
- Working with inheritance, instanceof, and casting
- Programming according to specifications

## Introduction

2024 is a leap year and nobody is more excited about this extra day than Freddy the Frog. Freddy loves to leap around on lily pads in his favourite ponds, and leap year means he gets an extra day to leap around and eat some flies before returning home to his mate, Franny. When Freddy is leaping around the ponds, he has to be careful of Allie the Alligator and her family. The alligators will eat Freddy if he gets too close. Thankfully, there are patches of reeds in some of the ponds in which Freddy can swim through without being seen by the alligators. However, this is only wise to do in dire situations in which he has no other way around the gators. There are also mud areas which Freddy cannot enter under any circumstances – he is allergic to mud.

The ponds are broken down into hexagonal cells in which each cell is of a specific type (see the list of Cell Types below). Freddy begins on a start cell, which is treated as a lily pad cell, and he can move from there one cell at a time toward the end cell, where Franny is. Freddy's movements are based on a variety of factors including the type of cell he is currently on and the types of cells in the region around the current cell, as explained below. Figure 1 shows an example of a pond with several types of cells.

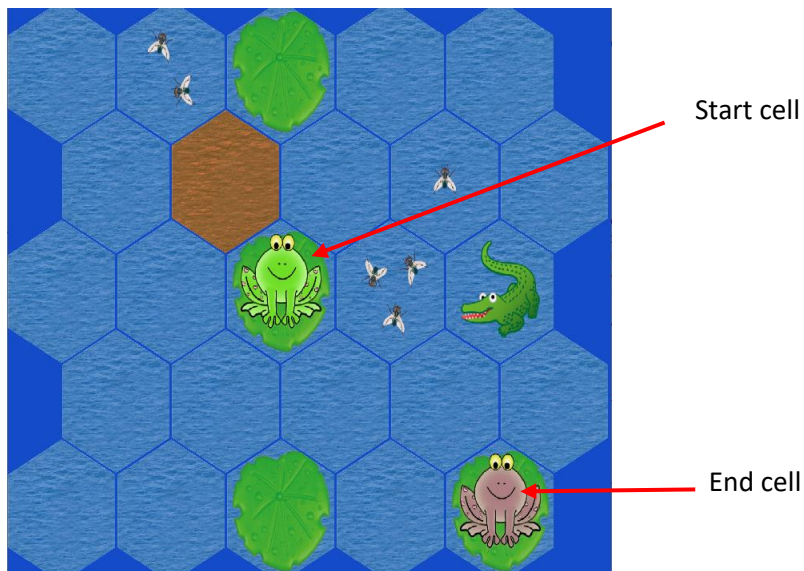


Figure 1. An example of a Pond with a variety of cells.

# Assignment 3

CS 1027



## Computer Science Fundamentals II

The ponds are always shown with the top being North. Based on this compass orientation, the hexagonal cells are connected such that each cell will have neighbouring nodes at North-East (index 0), East (1), South-East (2), South-West (3), West (4), and North-West (5). However, cells that are along an edge will have fewer than six neighbouring cells (they could have as little as two non-null neighbouring cells, depending on where on the edge they are located) but the same indexing will occur; there will just be one or more cells that are null. Figure 2 below illustrates the indexing system for both an internal cell and an edge cell – the cells shown with a yellow border. Note that the edge cell shown below has four non-null neighbours; the cells at index 0 and 5 are null.



Figure 2. A diagram illustrating the neighbouring indices from an internal cell (left) and from an edge cell (right).

## Cell Types

<b>Start</b>  This is where Freddy starts in the pond. This is also a lilypad cell.		
<b>End</b>  This is where Freddy's mate, Franny, is resting. This is also a lilypad cell. Freddy always tries to end here.		

# Assignment 3

CS 1027

## Computer Science Fundamentals II


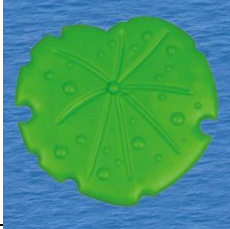
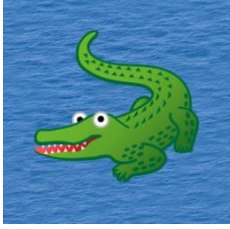
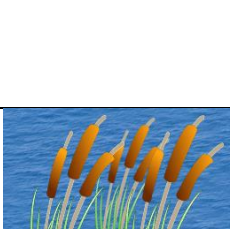
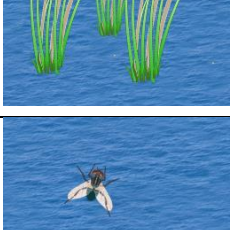
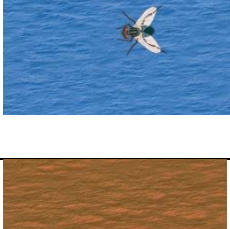
<b>Water</b>  This is a regular cell in which Freddy can swim. Nothing special happens here.		
<b>Lilypad</b>  Freddy loves lilypads because he can jump off them and can propel himself further than he can when swimming in water.		
<b>Alligator</b>  An alligator will eat Freddy if he gets too close. Hence, Freddy cannot jump to a cell with an alligator, or to a cell adjacent to an alligator unless that cell has reeds. For example in Figure10 below, Freddy could not go to cell 13, but he could go to cell 18.		
<b>Reeds</b>  Reeds are similar to water cells except that they can protect Freddy from being seen by an alligator located in an adjacent cell.		
<b>Flies (1, 2, or 3)</b>  These cells are similar to water cells except that they contain 1, 2, or 3 flies and Freddy will eat those flies when he swims through these cells.		
<b>Mud</b>  Freddy is allergic to mud. He cannot go on any mud cell at any time. He is allergic and gets swimmer's itch if he touches mud!		

Figure 3. A table illustrating and explaining each of the types of cells.

# Assignment 3

## What you Need to do

In this assignment you need to design a program that will allow Freddy to move from the starting cell to the end cell by following the movement rules specified below.

Generally, Freddy can just move from a cell to one of the six (or fewer) cells adjacent to the current cell. However, when Freddy is on a lilypad, he can also take a big leap to propel himself two cells away. He can **only** jump this far from a lilypad cell (note that his starting cell is always going to be treated as a lilypad cell). Freddy still prefers a shorter hop (i.e. landing in a cell adjacent to a lilypad) but may do the longer leaps depending on what kinds of cells are around him. Also, when he is doing a longer leap to two cells away, he prefers to leap to a cell that is two away **in a straight line**, meaning he's jumping over a whole hexagon to arrive at his destination cell; however, he can jump also two cells away to a cell that is not in a straight line from the current cell. These jumps and Freddy's preferences are illustrated in Figures 4, 5, and 6, and explained, along with additional information on movement and cell preferences, in the Movement Rules section.



Figure 4. The six (or fewer) adjacent cells which Freddy would prefer to jump to.



# Assignment 3

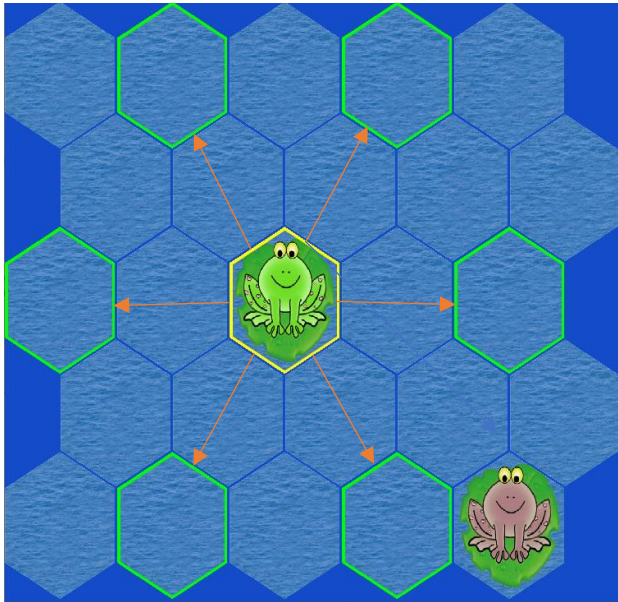


Figure 5. The six (or fewer) cells that are two away in a straight line (jumping over a whole hexagon cell to get to the cell). After adjacent cells, Freddy prefers to jump to a cell that is two away **in a straight line**.



Figure 6. The six (or fewer) cells that are two away **not** in a straight line (jumping over an edge, not a whole hexagon cell to get to the cell). These cells are the least preferred by Freddy.

# Assignment 3

## Movement Rules

As Freddy moves from the start cell toward the end cell he will always select to go from the current cell to the best next cell. To determine the next best cell for Freddy, we will assign priorities to the cells based on their location and type. So, you must first search through the six (or fewer) cells adjacent to the current cell and determine their priorities as explained below. Then, **if the current cell is a lilypad cell**, look at the cells two away from the current cell and determine their priorities. While examining the neighbouring cells around a cell, you must start at index 0 and continue clockwise up to index 5.

In Figure 7 below, the cells that are reachable from Freddy's current cell are outlined in green and labelled with a number. The number represents the order in which that cell must be considered. The numbers 0 through 5 are in red font because they represent the ring of adjacent cells which are most preferred. After they are examined, the cells two away from the current cell must be considered. Start with the neighbours of the cell labelled 0 in clockwise order (the cell labelled 6, then the cell labelled 7, next the cell labelled 1 (already considered, so it is not considered again), followed by the current cell (already considered), then the cell labelled 5 (already considered), and finally the cell labelled 8). Then, look at the neighbours of the cell labelled 1 in clockwise order. Continue doing this with the neighbours of all six (or fewer) cells that are adjacent to the current cell.

The order in which you check the cells is important because if there are two or more cells with the same priority, the one found first must take precedence over the others.



Figure 7. A diagram depicting the order in which to examine each of the cells around a lilypad cell. First, all adjacent cells must be examined (numbers shown in red). Then, cells two away must be examined by looking at the neighbours of each of the six adjacent cells.

# Assignment 3

## CS 1027 Computer Science Fundamentals II

As each cell is being examined, you must determine its priority based on its type and its position relative to the current cell. An initial priority is given based on the cell type, but it can be adjusted by 0.5 or 1.0 if the cell is not adjacent to the current cell. If the cell is non-adjacent to the current cell, but it is in a straight line from it (as depicted in Figure 5), the priority must be increased by 0.5. If it's non-adjacent and **not** in a straight line (as depicted in Figure 6), the priority must be increased by 1.0. Figure 8 shows a table with the priorities for each of the cell types as well as possible added weights for non-adjacent cells.

For example, a water cell that is adjacent to Freddy has a priority of 6.0. A water cell that is two cells away from Freddy in a straight line has a priority of 6.5. A water cell that is two cells away and not in a straight line has a priority of 7.0. These adjustments of 0.5 or 1.0 mean that those cells are less preferred than adjacent cells, however they could still have lower overall priorities than a cell adjacent to where Freddy is. For example, consider the pond in Figure 9. We examine the adjacent cells to the cell where Freddy currently is and get their priorities (each one of these cells has a priority of 6.0). Then we examine the cells that are cells 2 away in a straight line and get their priorities (6.5 each). Then we examine the cells that are 2 away not in a straight line, most of which have priorities of 7.0, except for the cell with flies which has a priority of 1.0 (3 flies means a priority of 0.0 but being 2 away in a non-straight line adds 1.0 to the priority, resulting in a total priority of 1.0). Hence, Freddy will jump immediately to the cell with the 3 flies since its total priority is lower than the priorities of all other reachable cells from his current cell.

Priority	Description
0.0	3 flies
1.0	2 flies
2.0	1 fly
3.0	End (Franny)
4.0	Lilypad
5.0	Reeds
6.0	Water
10.0	Reeds near alligator
+0.5	Cell 2 away in a straight line
+1.0	Cell 2 away not in a straight line

Figure 8. A table showing the priorities for the different cell types and possible adjustments to the priorities for non-adjacent cells.

As you examine each cell in the order explained and illustrated above, you must determine the priority of the cell based on its type (water, reeds, lilypad, etc.) as well as where it is located relative to the current cell (adjacent, two away in a straight line, or two away not in a straight line). However, when Freddy visits a cell, the cell will be marked and when looking for the cell with lowest priority **the cells that have been already marked will not be considered.**

# Assignment 3

## CS 1027 Computer Science Fundamentals II

Hence, to determine where Freddy will move next you must select the **unmarked** cell with the lowest priority, i.e. the best unmarked cell, as the one to visit next. If there are no possible cells to visit next, then nothing can be selected. If a next cell is selected then Freddy moves there and then the process is repeated until Freddy reaches the cell where Franny is located or until no more cells can be selected.

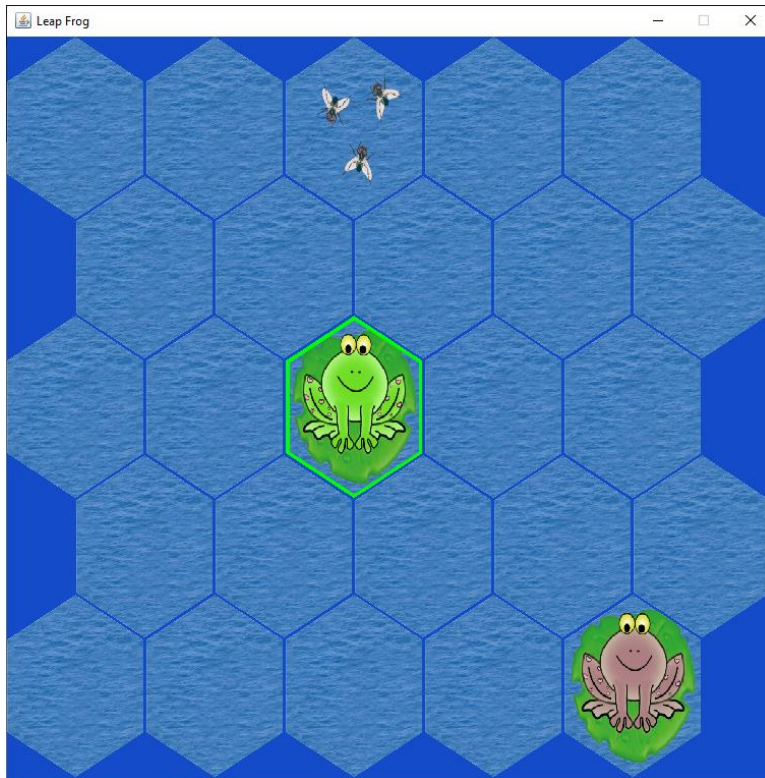


Figure 9. A simple pond in which Freddy will jump two away in a non-straight line rather than an adjacent cell, because there are 3 flies up there which makes that cell more desirable despite the longer jump.

You must use a Priority Queue (explained below) to store each of the possible cells that can be reached from the current cell when determining the best cell to visit next. Each cell will be represented with an object of the provided class Hexagon. The Priority Queue must store Hexagon objects and a priority value as calculated from the explanation and illustrations above. The Priority Queue cannot contain the same Hexagon object more than once; this is called a Unique Priority Queue.

## Priority Queue

A Priority Queue is an abstract data type which stores items ordered based on a given priority. When items are added into a Priority Queue, they must be placed in the correct order based on their priorities so that the item with the lowest priority is at the front, and each subsequent item has a priority greater (or equal\*) to the item before it in the collection.



# Assignment 3

## CS 1027 Computer Science Fundamentals II

\*If two or more items have the same priority, they must maintain the order in which they were added, i.e. a new item that has a priority of 5.0 will go **after** an existing item that has a priority of 5.0.

### Example

Suppose we have a Priority Queue and add each of the following items with their given priorities.

- "A", 3.0
- "B", 9.0
- "C", 7.0
- "D", 3.0
- "E", 5.0

The resulting Priority Queue would contain the following data items in this order:

---

A D E C B

---

A **Unique** Priority Queue cannot contain duplicate values. If we try to add an element that already exists in the Priority Queue, it should not add it again. Nothing should occur in this case.

### Example

Suppose we have a Unique Priority Queue and add each of the following items with their given priorities.

- "A", 3.0
- "B", 9.0
- "C", 7.0
- "D", 3.0
- "A", 1.0

The resulting Priority Queue would contain the following data items in this order:

---

A D C B

---

Notice the second time we try to add "A", nothing happens (regardless of its priority).

## Provided Files

There are many files provided to you for this assignment. Many of them are there to help set up the GUI (graphical user interface) or for custom exceptions, and you do not need to familiarize yourself with those files. However, there are some files that you should read through carefully to understand what is being done and how they are supposed to be used within the whole program. The primary files you should study in detail are Pond, UniquePriorityQueueADT, Hexagon, and FoodHexagon.

- StackADT.java
- ArrayStack.java
- Pond.java
- CellComponent.java
- HexLayout.java
- Hexagon.java
- FoodHexagon.java
- CollectionException.java
- IllegalArgumentException.java
- InvalidMapException.java
- InvalidNeighbourIndexException.java
- UniquePriorityQueueADT.java
- TestUPQ.java
- TestPath.java
- TestSetup.java

TestUPQ.java and TestPath.java are tester files to **help** check if your java classes (ArrayUniquePriorityQueue and FrogPath, respectively) are implemented correctly. Class TestSetup helps you check that you placed all required files in the correct location.

Similar tester 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.**

## Classes to Implement

For this assignment, you must implement two Java classes: **ArrayUniquePriorityQueue** and **FrogPath**. 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).

# Assignment 3

## CS 1027 Computer Science Fundamentals II

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.PriorityQueue`.

### ArrayUniquePriorityQueue.java

This class must implement the provided `UniquePriorityQueueADT` interface and must support the generic type `T`. Since it implements this interface, it must include all methods that are defined in the interface. It must use arrays as the underlying data structure to implement this ADT. More specifically, it must use two arrays: `queue` of type `T` to store the data items and `priority` of type `double` to store the priorities that correspond to the data items. These two arrays must relate to one another element-wise, i.e. `queue[0]` has a priority of `priority[0]`, `queue[1]` has a priority of `priority[1]`, and so on. Instance variable `count` specifies the number of data items in the Unique Priority Queue.

The class must have the following private instance variables:

- private `T[] queue`
- private `double[] priority`
- private `int count`

The class must have the following public methods:

- public `ArrayUniquePriorityQueue ()`: constructor
  - Initialize both arrays with capacities of 10 (this can be later expanded if needed) and set `count = 0`
- public void `add (T data, double prio)`
  - Check if the given `data` item is already contained anywhere in `queue`, and if so, end the method immediately without doing anything.
  - If the arrays are full (i.e. there is no space for the new item), then expand the capacity of both arrays by adding 5 additional spaces to each array.
  - Add the given `data` item into the `queue` such that the priorities are ordered from lowest to highest \*\*. Find the correct index `i` in the `priority` array where `prio` should be placed to maintain the proper ordering and place `prio` in array `priority` at index `i`. Likewise, place `data` in array `queue` at index `i`. Shift any subsequent values from both arrays over to the right to make room for this new item. Increment `count`.
  - \*\* If the new item has the same priority as an existing item, the new item must go **after** the existing item of the same priority. For example, if you add "green" with priority 4.0 and then add "purple" with priority 4.0, you must add "purple" after "green" since it was added after.
- public boolean `contains (T data)`

# Assignment 3

## CS 1027 Computer Science Fundamentals II

- Return true if the given *data* item is contained somewhere in *queue*, or false otherwise.
- public T peek () throws CollectionException
  - If the priority queue is empty, throw a CollectionException with the message "PQ is empty".
  - If it is not empty, return the item with the smallest priority without removing or changing it.
- public T removeMin () throws CollectionException
  - If the priority queue is empty, throw a CollectionException with the message "PQ is empty".
  - If it is not empty, remove and return the item with the smallest priority. Shift any subsequent values from both arrays over to the left to fill the gap made by the removed item. Decrement *count*.
- public void updatePriority (T *data*, double *newPrio*) throws CollectionException
  - Check if the given *data* item is already contained anywhere in *queue*, and if **not**, throw a CollectionException with the message "Item not found in PQ".
  - Update the priority of the existing *data* to a value of *newPrio*. Ensure that *data* and its new priority are re-located within *queue* and *priority* so that the order of the priorities is maintained after the update.
  - Hint: it may be simpler to remove the existing item and its priority and then re-add it with the new priority.
- public boolean isEmpty ()
  - Return true if the priority queue is empty, or false otherwise.
- public int size ()
  - Return the number of items stored in the priority queue.
- public int getLength ()
  - Return the capacity of the arrays (they should both have the same capacity at all times).
  - Note that this method is not part of the UniquePriorityQueueADT but still a required method as it will be used for testing purposes.
- public String toString ()
  - If the priority queue is empty, return the string "The PQ is empty."
  - If it is not empty, create a String that contains each data item from the queue followed by its corresponding priority in square brackets. The String must be formatted as shown here:

yellow [2.0], blue [3.0], red [5.0], green [8.0], purple [9.0]

### FrogPath.java

In this class, you must determine the path that Freddy the Frog must follow to go from the starting cell to the end cell, where Franny is, using the given movement rules described above.

The class must have the following private instance variable:



# Assignment 3

## CS 1027 Computer Science Fundamentals II

- private Pond *pond*

The class must have the following public methods:

- public FrogPath (String *filename*): constructor
  - Initialize pond using the given filename by invoking the constructor of the provided class Pond passing as parameter *filename*.
  - Surround this initialization line in a try-catch structure since the Pond constructor may throw checked exceptions.
- public Hexagon findBest(Hexagon *currCell*)
  - Create a PriorityQueue to store all unmarked cells that can be reached from *currCell*. Compute the priorities for the cells based on the order in which the cells should be chosen. See the Movement Rules section above for more information on this order. Use method isMarked() from class Hexagon to determine whether a cell has been marked. You will also need to use methods isMudCell(), isAlligator(), isReedsCell(), isEnd(), isLilyPad(), and isWaterCell() from class Hexagon, and method getNumflies() from class FoodHexagon.
  - If there is at least one item in the priority queue, return the item with the lowest priority (i.e. the item at the front). If the priority queue is empty, return null.
- public String findPath()
  - Create a Stack using the provided ArrayStack class to keep track of the cells that the frog has visited in its path from the starting cell toward the end cell. Remember that the cells are represented with objects of the class Hexagon. See the Path Algorithm section below for the steps that your program needs to perform to compute this path.
  - You also need to build a String containing the cell ID's (call getID() or toString() method on the Hexagon objects to get a the ID of a cell) of **EVERY** cell Freddy visits along his path. Cell ID's must be separate by spaces in this string. If a solution is found, add "ate N flies" (where N is the number of flies he ate) to the end of this string and return this string. If no solution is found (i.e. it is impossible for him to reach the end cell), return the string "No solution" instead. Note that if the path followed by Freddy visits the same cell A several times, then the String will contain multiple occurrences of this cell A.  
Figure 10 below shows an example of a pond and the path that the frog follows from the start to the end cell. The String that method findPath() must produce for this example is  
12 1 2 3 4 3 7 3 2 11 16 22 24 ate 2 flies  
Note that in the above String cell 2 appears twice and cell 3 appears 3 times. In Figure 10 the cells that appear more than once in the String are shown in red. Even though a cell is marked the first time that Freddy visits it, a cell might appear multiple times in the String returned by this method because Freddy might have to return to a previously visited cell if the path that he selected does not lead to the end cell. To better understand this, please study the path algorithm given in the next section.
- public static void main (String[] *args*)

CS 1027  
Computer Science Fundamentals II

- 

## Path Algorithm

```
Initialize stack S
Push start cell onto S and mark start as in-stack
fliesEaten = 0

Create an empty string
while S is not empty {
    curr = S.peek

    Add the ID of curr to the string
    if curr is the end cell
        end loop immediately
    if curr is a food (fly) cell {
```

# Assignment 3

## CS 1027 Computer Science Fundamentals II

```
        fliesEaten = fliesEaten + num of flies on cell
        remove flies from cell (call removeFlies() on the cell)
    }
    next = findBest(curr)
    if next = null {
        S.pop
        mark curr as out-of-stack
    } else {
        S.push(next)
        mark next as in-stack
    }
}
```

Check if the stack is empty and either change the string to “No solution” or add to it the number of flies that the frog ate.

**Notes.** Use method `getStart()` from class `Pond` to get the start cell. Use methods `markInStack()` and `markOutStack()` of class `ArrayStack` to mark items in the stack. You can also use methods `isEmpty()`, `peek()`, `push()` and `pop()` from class `ArrayStack`. Use operator `instanceof` to check if the current cell is an object of class `FoodHexagon`. You might also need to use casting to allow variables of type `Hexagon` to make reference to objects of the class `FoodHexagon`. Method `removeFlies()` is from class `FoodHexagon`.

## Command Line Arguments

From the terminal you can run the program by first placing all the files in the same directory and then compiling it by typing:

```
javac FrogPath.java
```

then, run the program by typing

```
java FrogPath filename
```

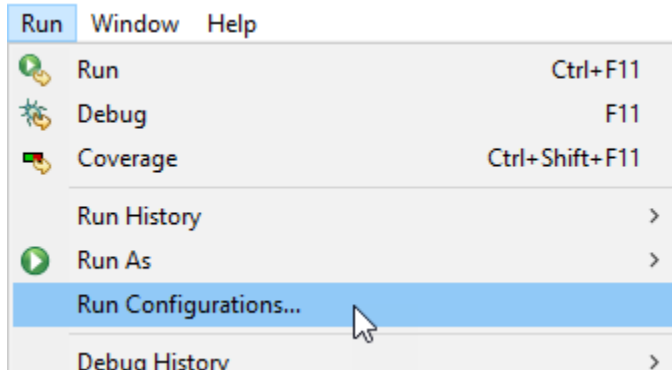
where filename is the name of the input file. You need to add a *main* method in this class to be able to run the program. Several input files (called pond1.txt, ..., pond9.txt) are provided.

In Eclipse, you must place all the image files and txt files in the project root directory (the directory containing the src directory; do not place the image and txt files in the src directory as then Eclipse will not be able to find them). You set the command line arguments by following these steps.

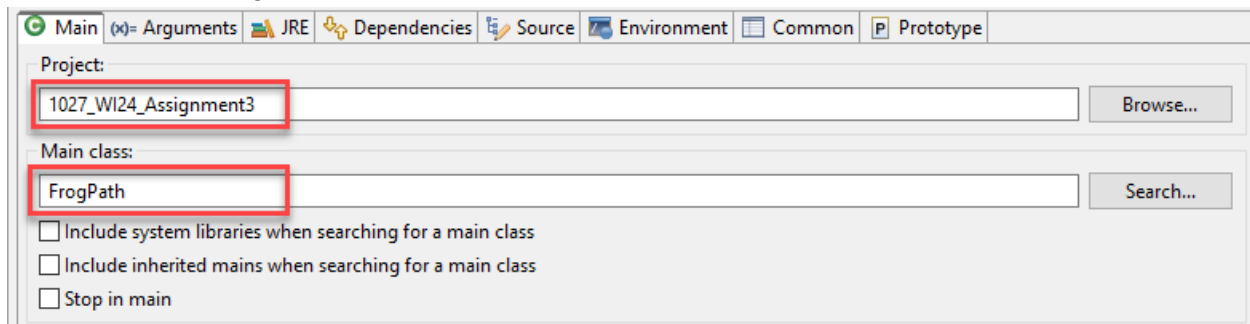
# Assignment 3

## CS 1027 Computer Science Fundamentals II

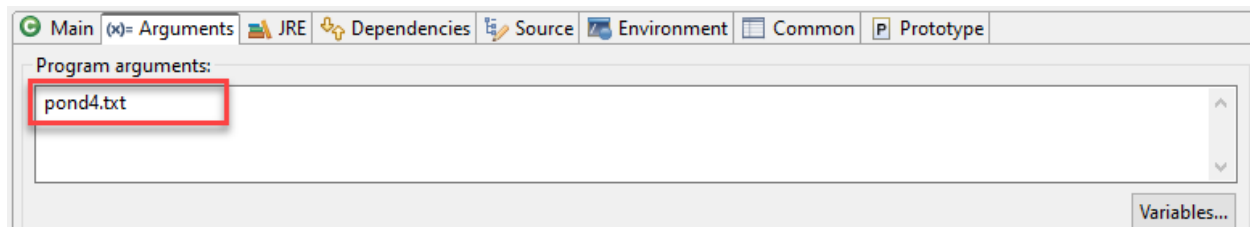
1. At the top, click on Run and select Run Configurations...



2. In the Run Configurations window, make sure you have the correct project selected and choose FrogPath as the Main class.



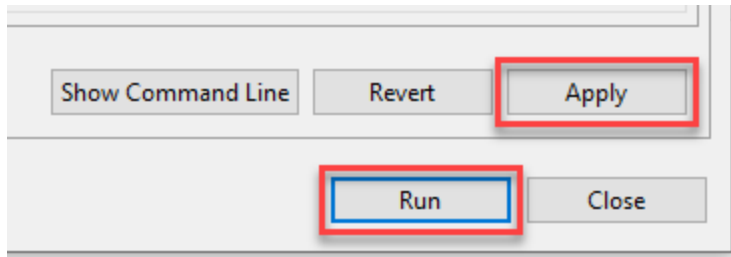
3. Click the "Arguments" tab and type in your argument, which should be the filename of one of the pond text files.





# Assignment 3

- Click Apply and then Run.



The argument(s) provided in the command line arguments will be sent in to the main method as the String array "args" parameter. Within your FrogPath class, you should include a main method like the one provided below so you can test your algorithm on a single pond, specifically the one whose name is given in the command line argument.

```
public static void main (String[] args) {  
    if (args.length != 1) {  
        System.out.println("No map file specified in the arguments");  
        return;  
    }  
    FrogPath fp = new FrogPath(args[0]);  
    Hexagon.TIME_DELAY = 500; // Change this time delay as desired.  
    String result = fp.findPath();  
    System.out.println(result);  
}
```

## 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)?

# Assignment 3

## CS 1027 Computer Science Fundamentals II

- 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.

**Submission** (due Wednesday, March 20 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

- ArrayUniquePriorityQueue.java
- FrogPath.java

### Grading Criteria

# Assignment 3

CS 1027  
Computer Science Fundamentals II

Total Marks: [20]

## Functional Specifications:

[2] ArrayUniquePriorityQueue.java

[3] FrogPath.java

[13] Passing Tests (some additional, hidden tests will be run on Gradescope)

## Non-Functional Specifications:

[0.5] Meaningful variable names, private instance variables

[0.5] Code readability and indentation

[1] Code comments