Assignment 2

COMP 250 Winter 2023

posted: Friday, March 3, 2023
due: Monday, March 20, 2023 at 11:59m

Learning Objectives

This assignment is meant for you to practice different data structures and their applications. A lot of the design decisions have been done for you, but you might notice that the instructions for A2 are less complete than A1. You will need to take care of small details by yourself. Finally, we will not be grading your work on efficiency; however, if a method takes more than 1 second to complete, it is equivalent to failing.

General Instructions

This assignment contains 3 parts. You need to download the files provided and complete the following 13 files:

Part I	Part II	Part III

`MyList.java`
`MyLinkedList.java`
`MyDoublyLinkedList.java`
`MyStack.java`
`MyQueue.java`
	`Position.java`
	`TargetQueue.java`
	`Direction.java`
	`ActionQueue.java`
		`Region.java`
		`Caterpillar.java`
		`State.java`
		`World.java`

- Except for the `MyDoublyLinkedList.java`, where a starting file is provided, you have to create all the above classes from scratch.
- Please make sure that all the files you submit are part of a package called `assignment2`.
- You are NOT allowed to import any other class/interface except `NoSuchElementException` and `Iterator`, which have already been imported for you in `MyDoublyLinkedList.java`. You are also not allowed to use ANY field of type array. **Any failure to comply with these rules will give you an automatic 0.**
- You are always allowed to add `private` methods/fields whenever you think this can improve your implementation. You are also allowed to override the `toString` method if this can help you during the debugging process.
- Two extra files, `CaterpillarGame.java` and `CaterpillarDrawer.java`, are provided for visualizing the results of Part III. Do not use them unless your Part III is completed and passes all tests from the minitester.

- Do NOT start testing your code only after you are done writing the entire assignment. It will be extremely hard to debug your program otherwise. If you need help debugging, feel free to reach out to the teaching staff. When doing so, make sure to mention what is the bug you are trying to fix, what have you tried to do to fix it, and where have you isolated the error to be.

Submission instructions

- Late assignments will be accepted up to 2 days late and will be penalized by 10 points per late day. Note that submitting one minute late is the same as submitting 23 hours late. We will deduct points for any student who has to resubmit after the due date (i.e. late) irrespective of the reason, be it the wrong file submitted, the wrong file format submitted or any other reason. We will not accept any submission after the 2 days grace period.
- Don't worry if you realize that you made a mistake after you submitted: you can submit multiple times but **only the latest submission will be evaluated**. We encourage you to submit a first version a few days before the deadline (computer crashes do happen and Ed Lessons may be overloaded during rush hours).
- **Do not submit any other files, especially .class files and the tester files. Any failure to comply with these rules will give you an automatic 0.**
- Whenever you submit your files to Ed, **you will see the results of some exposed tests**. If you do not see the results, your assignment is not submitted correctly. **If your assignment is not submitted correctly, you will get an automatic 0. If your submission does not compile on ED, you will get an automatic 0.**
- The assignment shall be graded automatically on ED. **Requests to evaluate the assignment manually shall not be entertained, and it might result in your final marks being lower than the results from the auto-tests.** Please make sure that you follow the instruction closely or your code may fail to pass the automatic tests.
- The exposed tests on ED are a mini version of the tests we will be using to grade your work. If your code fails those tests, it means that there is a mistake somewhere. Even if your code passes those tests, it may still contain some errors. Please note that these tests are only a subset of what we will be running on your submissions, we will test your code on a more challenging set of examples. Passing the exposed tests assures you that your submission will not receive a grade lower than 40/100. We highly encourage you to test your code thoroughly before submitting your final version.
- Next week, a mini-tester will also be posted. The mini-tester contains tests that are equivalent to those exposed on Ed. We encourage you to modify and expand it. ***You are welcome to share your tester code with other students on Ed.*** Try to identify tricky cases. Do **not** hand in your tester code.
- **Failure to comply with any of these rules will be penalized**. If anything is unclear, it is up to you to clarify it by asking either directly a TA during office hours, or on the discussion board on Ed.

PART - I: Data Structures (35 points)

For the first part, you will write several classes to implement different data structures.

[5 points] Write an interface `MyList<E>` with a generic element type E. The `MyList<E>` interface extends the `Iterable<E>` interface. The `MyList<E>` interface has the following `public` methods:

- A `getSize` method that returns an `int` indicating the number of elements in the list.
- A `isEmpty` method that returns a `boolean` indicating whether the list is empty.
- An `add` method that appends the specified element to the end of this list and does not return anything.
- A `clear` method that removes all of the elements from this list and does not return anything.
- A `remove` method that removes the last element from the list and returns it.

[5 points] Write an abstract class `MyLinkedList<E>` with a generic type `E` and implements `MyList<E>`.

The `MyLinkedList<E>` class has the following `protected` field:

- A `int` indicating the size of the list.

The class must also implement the `public` methods `getSize` and `isEmpty` as specified by the documentation of the interface `MyList<E>`.

[15 points] Complete the `MyDoublyLinkedList<E>` class (provided) as a doubly linked-list of generic type `E`. Inside the file provided you can find:

- A private class called `DLLIterator` and a method `iterator` returning a `DLLIterator` object. This `DLLIterator` class implements the `Iterator<E>` interface (from `java.util.Iterator`) which allows the user to iterate over the list. DO NOT modify anything from the `DLLIterator` class or the `iterator` method; otherwise, your program will fail the testers.
- A `private` class called `DNode`. The `DNode` class contains three `private` fields:
  - An `E` named `element` indicating the element stored in the node
  - A `DNode` named `next` indicating the next node in the list
  - A `DNode` named `prev` indicating the previous node in the list
- The `MyDoublyLinkedList<E>` class has the following `private` fields:
  - A `DNode` named `head` indicating the head of the list
  - A `DNode` named `tail` indicating the tail of the list

It is up to you to do the following:

- Make sure the class `MyDoublyLinkedList<E>` extends the `MyLinkedList<E>` class.
- Implement the `public` methods `add`, `remove`, and `clear` as specified in the documentation of the interface `MyList<E>`.
- Add the following `public` methods to the `MyDoublyLinkedList<E>` class:
  - An `addFirst` and an `addLast` method which takes input as an object of the generic type `E` and adds the element as the first/last element of the list. These methods do not return anything.
  - A `removeFirst` and a `removeLast` method which removes the first/last node from the list and returns the element. The method should throw a `NoSuchElementException` if the list is empty.
  - A `peekFirst` and a `peekLast` method which returns the first/last element without removing it from the list. The method should throw a `NoSuchElementException` if the list is empty.
  - An `equals` method that takes an `Object` as an input; the method returns `true` if all elements in the list are considered to be *equal* and `false` otherwise.

You are free to add any additional `private` methods and/or fields to either `MyDoublyLinkedList<E>` or `DNode` if you think this can help you with the implementation. Please note that, as specified in the general instructions, you are NOT allowed to add any field of type array.

[5 points] Write a class `MyStack<E>`.

The `MyStack<E>` class has the following `private` fields:

- A `MyDoublyLinkedList<E>` to store the elements of the stack.

The class must also have the following `public` methods:

- A constructor that takes no inputs and creates an empty stack.
- A `push` method that takes an input of type `E` and adds the input to the top of the stack.
- A `pop` method that removes the top of the stack and returns it. The method should throw a `NoSuchElementException` if the stack is empty.
- A `peek` method that returns the element at the top of the stack without removing it. The method should throw a `NoSuchElementException` if the stack is empty.
- A `isEmpty` method that returns true if the stack is empty and false otherwise.
- A `clear` method that clears the stack (i.e. removes all the elements from the stack).
- A `getSize` method that returns the number of elements in the stack.

[5 points] Write a class `MyQueue<E>`.

The `MyQueue<E>` class has the following `private` fields:

- A `MyDoublyLinkedList<E>` to store the elements of the queue.

The class must also have the following `public` methods:

- A constructor that takes no inputs and creates an empty queue.
- A `enqueue` method that takes an input of type `E` and adds it to the back of the queue.
- A `dequeue` method that removes the first element from the front of the queue and returns it. The method should throw a `NoSuchElementException` if the queue is empty.
- An `isEmpty` method that returns true if the queue is empty and false otherwise.
- A `clear` method that clears the queue (i.e. removes all the elements from the queue).
- An `equals` method that takes as input an `Object` and returns true if the input is another queue with equal elements, and false otherwise.

PART - II: Information Parsing with Stacks (35 points)

In PART-II, you will decode and parse strings that contain compressed information.

[7 points] Write a `Position` class as a representation of a point `(x,y)` on a 2D map as the figure below. The origin `(0,0)` is located at the top-left corner of the map.

(0,0)	(1,0)	(2,0)	→
(0,1)	(1,1)	(2,1)	
(0,2)	(1,2)	(2,2)	
↓			

The `Position` class has the following `private` fields:

- An `int` indicating the x-coordinate on the map
- An `int` indicating the y-coordinate on the map

The `Position` class must also have the following `public` methods:

- A constructor that takes two `int` as inputs indicating the x- and y-coordinates on a map, in this order. The constructor uses the inputs to initialize the corresponding fields.
- A constructor that takes a `Position` as input and initializes the fields with the corresponding input. This type of constructors are called *copy constructors*. They are used to create duplicates of existing instances of the class.
- A `reset` method that takes two `int` as inputs indicating the x- and y-coordinates on a map. The method uses the inputs to resets the corresponding fields.
- A `reset` method that takes `Position` as input and resets the corresponding fields using the values of those of the input object.
- A `static` method called `getDistance` that takes two `Position` objects as input and returns the absolute distance between them in terms of the x- and y- coordinates. For example, the distance between two `Position` objects `(2,3)` and `(3,5)` is `3` (i.e. `|2-3| + |3-5|`).
- A `getX` and a `getY` method that return the values of the corresponding fields.
- A `moveWest` method that decrements the x-coordinate by 1.
- A `moveEast` method that increments the x-coordinate by 1.
- A `moveNorth` method that decrements the y-coordinate by 1.
- A `moveSouth` method that increments the y-coordinate by 1.
- An `equals` method which takes as input an `Object` and returns `true` if the input matches `this` in type, x-coordinate, and y-coordinate. Otherwise, the method returns `false`.

[13 points] Write a `TargetQueue` class that extends from `MyQueue<Position>`. The `TargetQueue` class is designed to store a list of `Position` objects. The `TargetQueue` can parse a string and store the information as a queue. For example, given a string `"(1,2).(3,4).(5,6)."`, the input should be parsed into a queue of 3 positions shown in the figure below:

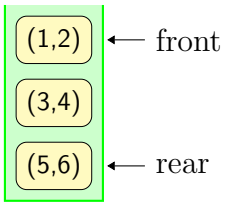

Figure 1: A queue of positions after parsing `"(1,2).(3,4).(5,6)"`

Note that, the input string has this strict format `(x,y)`. The x- and y-coordinates are enclosed by a left and a right parenthesis separated by a comma, and two positions must be separated by a period.

The `TargetQueue` class has the following `private` fields:

- A `MyStack<String>` to help with parsing a string containing information about the positions.

The `TargetQueue` class must also have the following `public` methods:

- A constructor that takes no input. The constructor should invoke the constructor of the superclass and initialize its fields.
- A `clear` method that clears this queue and its stack field.
- An `addTargets` method that takes input as a `String` and parses it into a queue of `Positions`. The method should throw an `IllegalArgumentException` if the syntax of the input is invalid. It can be useful for this method to have a local `String` variable to store the numbers contained in the input string. I will refer to this variable as `num`. To check the validity of the syntax and parse out the information regarding the positions, we need to iterate through every character in the input string.
  - if the character is a left parenthesis, the stack and `num` should both be empty. If so, push `"("` onto the stack.
  - if the character is a digit, append the digit to `num`.
  - if the character is a comma, check if `num` is a valid integer. If `num` is empty, we must have read a comma before the x-coordinate, which is a syntax error. Otherwise, push `num` and `","` onto the stack, in exactly this order. Finally, reset `num`.
  - if the character is a right parenthesis
    - the stack should contain 3 elements with exactly the order a comma, an integer, and a left parenthesis from the top. If the order is not correct or any of the elements are missing, the input string has a syntax error.

* `num` should be an integer representing the y-coordinate. If `num` is empty, we must have reached the right parenthesis without finding a y-coordinate, which is a syntax error.
* If the syntax is correct, add the x-y coordinate as a `Position` object onto the queue and reset `num`.

– if the character is a period, we must have reached the end of expression for a particular position. If the syntax is correct, the stack and `num` should both be empty. Otherwise, this is another syntax error.

Note that, these are the minimum cases of syntax errors that you should take care of. There might be more edge cases that do not get detected by the instructions above. It would be your job to find them and throw exceptions when needed.

[2 points] Write an enumeration `Direction`.

An enumeration in java is a special data type that represents a fixed set of constants. Defining and using enumerations will make your code a bit more readable. The syntax is as follows:

Defining an Enumeration	Using Enumeration

`public enum ENUM_NAME {`
`  LABEL_1,`
`  LABEL_2,`
`  ...`
`  LABEL_N`
`}`
	`ENUM_NAME e = ENUM_NAME.LABEL_1 ;`
	`ENUM_NAME f = ENUM_NAME.LABEL_2 ;`
	`...`
	`switch (e) {`
	`  case LABEL_1:`
	`  break ;`
	`  case LABEL_2:`
	`  break ;`
	`}`

Inside a file called `Direction.java` you should define an enumeration called `Direction` that represents the directions. The enumeration `Direction` has four values `NORTH, SOUTH, WEST, EAST`.

[13 points] Write a `ActionQueue` class that extends `MyQueue<Direction>`. The `ActionQueue` class decodes a string that contains compressed information on directions into a queue of `Direction` objects. The input strings are encoded in the form

$$K[D]$$

which indicates that the direction(s) D is repeated K times. K should be a positive number. D should contain characters `'N'`, `'S'`, `'W'`, `'E'` that represent directions. Specifically, `'N'` stands for `NORTH`, `'S'` stands for `SOUTH`, `'W'` stands for `WEST`, and `'E'` stands for `EAST`. Table 1 contains some examples of encoded strings and their decoded values:

Note that, implementing this class is meant to be a bit more challenging, and hence, the implementation details are omitted. You will need to understand and analyze the task at hand and decide how to go about implementing such class.

Encoded	Decoded
3[N]	NNN
3[NE]	NENENE
3[2[N]2[E]]1[S]	NNEENNEENNEES
2E[N]	Syntax Error: E before 2
EN]	Syntax Error: Missing [
E[EN]	Syntax Error: Missing K
A	Syntax Error: Unknown Character A

Table 1: Examples of encoded and decoded directions

It is up to you to decide how many, and what type of `private` fields you might need for this class. Different implementations are possible. Remeber to exploit the properties of stacks to help you parse strings.

The `ActionQueue` class must have the following `public` methods:

- A constructor that takes no input and takes care of initializing whichever field(s) you have added.
- A `clear` method that removes all items from this queue (and possibly clears the field(s))
- A `loadFromEncodedString` method that takes input as a `String` of an encoded message and converts it into a queue of `Direction`. Similar to the `TargetQueue`, the method should throw an `IllegalArgumentException` if the syntax of the input is invalid. It is up to you to figure out when and where to throw exception, or otherwise to translate the string into the corresponding queue of `Direction`s. I suggest you start by focusing on how to correctly parse very simple strings like `3[N]`. Then move onto correctly parsing sequences of simple strings like `2[N]3[W]` or more complicated strings such as `3[NE]`. Finally, try to solve the problem of parsing nested strings such as `3[2[N]2[E]]`. Please do not focus on getting this part to work to perfection before moving onto Part III of the assignment. You do not need this to be working to complete what comes next. In fact, you can still get a very high mark even if this part is only partially working. Make sure to use your time wisely!

PART - III: Caterpillar Game (30 points)

In PART-III, you will write a small caterpillar game. The figure below shows a 2D map with a caterpillar (blue) and food (red). The caterpillar can move around in this 2D map. Whenever the caterpillar's head reaches the food, it can "eat" the food and grow one cell bigger. You will write a small program that moves the caterpillar, and the final goal is to eat everything on the map.

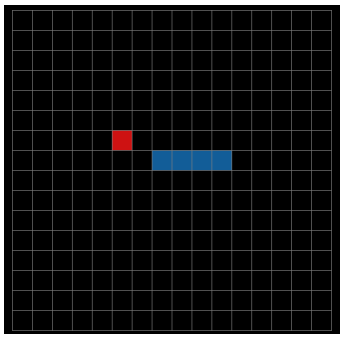

Figure 2: The caterpillar (blue) and the food (red) in a 2D map. The xy-coordinate of the top-left corner is (0,0), and the xy-coordinate of the bottom-right corner is (15,15).

[3 points] Write a `Region` class that represents a 2D world.

The `Region` class has the following `private` fields:

- A `int` indicating the minimum x-position
- A `int` indicating the minimum y-position
- A `int` indicating the maximum x-position
- A `int` indicating the maximum y-position

The `Region` class has the following `public` methods:

- A constructors with 4 inputs as `int` which indicates minimum x position, minimum y position, maximum x position, and maximum y position (in exactly this order).
- A `contains` method that takes an input as `Position`. The method returns true if the position is within the range of the `Region` and returns false otherwise.

For example, if the range is $x \in [0, 5], y \in [3, 6]$, then the minimum x and y positions are 0 and 3, and the maximum x and y positions are 5 and 6 (inclusive). The method `contains` should return true if the input is `(0,6)` and return false if the input is `(10,11)`.

[10 points] Write a `Caterpillar` class that extends `MyDoublyLinkedList<Position>`.

A `Caterpillar` is represented as a `MyDoublyLinkedList<Position>` that completely determines the positions of the body of a caterpillar within a 2D map. The `Caterpillar` class must have the following `public` methods:

- A constructor that takes no input and sets the initial state of the caterpillar at the beginning of a game. When the game starts, the initial size of the caterpillar should be 1 and its initial position is at `(7,7)`.
- A `getHead` method that returns a `Position` object representing the head position of the caterpillar.
- An `eat` method that takes as input a `Position` and adds the input to the front of the list, making this be the new position for the head of the caterpillar. If the input position is not orthogonally adjacent to the current head position, throw an `IllegalArgumentException`.
- A `move` method that takes as input a `Position`, adds the input to the front of the list, and removes one element from the back of the list. If the input position is not adjacent to the current head position, throw an `IllegalArgumentException`.
- A `selfCollision` method that takes as input a `Position` and returns true if the input overlaps with one of the body parts, and returns false otherwise.

[2 points] Write a `GameState` enumeration. The `GameState` enumeration contains the following labels: WALL_COLLISION, SELF_COLLISION, NO_MORE_ACTION, EAT, MOVE, DONE.

[15 points] Write a `World` class.

The `World` class should have the following `private` fields (You can add more private fields whenever fits):

- A `Caterpillar` that represents the caterpillar.
- A `Position` that indicates the current food position.
- A `Region` that represents the 2D map where the caterpillar can move.
- A `ActionQueue` that stores a list of actions to take for the caterpillar.
- A `TargetQueue` that stores a list of food positions that the caterpillar should travel to.
- A `GameState` which indicates the current state of the game.

The `World` class must also have the following `public` methods:

- A constructor that takes two input arguments; the first argument is a `TargetQueue` and the second argument an `ActionQueue`. The constructor uses the inputs to initialize the appropriate fields. The `Region` of the game is set as $x \in [0, 15], y \in [0, 15]$, and the caterpillar is initialized with its initial state (its size is 1, and it is positioned at $(7, 7)$). Before the game starts, dequeue the food position from the `TargetQueue`, and set the `GameState` to `MOVE`. You can assume that the game will start with at least one food to be eaten by the caterpillar.

- A `step` method that makes a step in the game. Here are the procedures:
  - First, take the next heading direction from the `ActionQueue`. If the `ActionQueue` is empty, set the `GameState` to `NO_MORE_ACTION`.
  - Second, if the `GameState` is not `MOVE` or `EAT`, `return`.
  - Then, get the current head position of the caterpillar and calculate the next head position assuming it is moving according to the heading direction. For example, if the head position of the caterpillar is (5,7) and the direction is `EAST`, the next position will be (6,7). Depending on the next position calculated above, You will need to handle the following scenarios:
    * If moving to the next position will result in the caterpillar moving out of the map, set the `GameState` to `WALL_COLLISION`.
    * If moving to the next position will result in a self-collision of the caterpillar, set the `GameState` to `SELF_COLLISION`.
    * If the food is located at the next position, the caterpillar can "eat" it. If there is not more food on the `TargetQueue`, the caterpillar must have eaten everything, and we can set the `GameState` to `DONE`. Otherwise, dequeue the next food position from the `TargetQueue` and set the `GameState` to `EAT`.
    * Otherwise, move the caterpillar to the next position and set the `GameState` to `MOVE`.
- A `getState` method that returns the current state of the game as a `GameState`.
- A `getCaterpillar` method that returns the caterpillar as a `Caterpillar` object.
- A `getFood` method that returns the food position as a `Position` object.
- An `isRunning` method that returns true if the game is still running (`MOVE` or `EAT`) or false if the game is over.

[0 point] The Graphical Interface. You are provided with a graphical interface for visualizing the movement of the caterpillar (`CaterpillarGame.java` and `CaterpillarDrawer.java`).

- `CaterpillarGame.java` and `CaterpillarDrawer.java` should be under package `game`.
- You can visualize the game by running `CaterpillarGame.java`. You are more than welcome to change the targets and directions as you like to test if your game is running correctly. You do not need to understand the implementation of these two files.
- Do not upload `CaterpillarGame.java` and `CaterpillarDrawer.java` on ED. **Failing to do so will result in an automatic 0.**