COMPUTER SCIENCE 1027B – Foundations of Computer Science II

Assignment 3

Due Date: March 19, 11:55 pm

1. Purpose

To gain experience with

- Linked structures and doubly linked lists
- Exception handling
- Algorithm design and modular design.

2. Introduction

For this assignment you will write a program that plays a variation of the snake game that we considered in the first assignment. This time there when playing the game there will be a set of eagles chasing the snake, and if an eagle gets to the snake, the snake will die. The goal, as before, is for the snake to eat as many apples as possible while avoiding being killed. As in Assignment 1, the game board consists of a set of square tiles, where objects are positioned and through which the snake and the eagles can move.

The methods that you must implement for this assignment are similar to those for assignment 1, however this time you will not store the game board in a matrix or the snake in an array, instead you will use linked structures. The snake will be stored in a doubly linked list and the game board will be stored in an array of doubly linked lists.

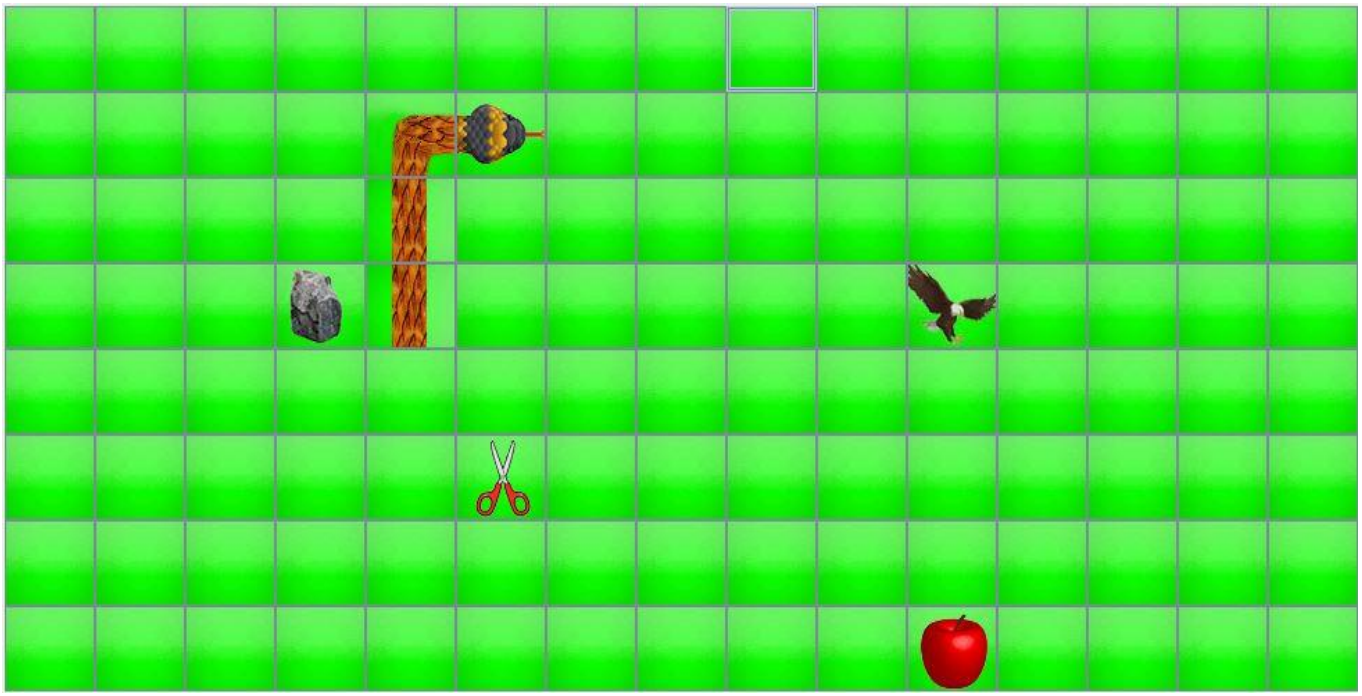

Figure 1. A gameboard of 8 rows and 15 columns containing one apple, one rock, one pair of scissors, an eagle, and a snake of length 4.

3. Classes to Implement

For this assignment you need to implement 5 Java classes: *Position*, *DoubleNode*, *DoubleList*, *SnakeLinked*, and *BoardGameLinked*. Your program must catch any exceptions that might be thrown. For each exception caught an appropriate message must be printed. The message must explain what caused the exception to be thrown.

3.1 Class *Position*

This is the same class as in Assignment 1.

3.2 Class *DoubleNode*

This class represents the nodes in a doubly linked list. The header of this class must be this:

public class DoubleNode<T>

This class will have the following instance variables:

- *private DoubleNode<T> next*. A reference to the next node in the list.
- *private DoubleNode<T> prev.* A reference to the previous node in the list.
- *private T data.* The data stored in this node.

In this class you must implement the following methods:

- *public DoubleNode ().* Creates an empty node, where all instance variables are null.
- *public DoubleNode (T newData).* Creates a node storing the given data in which *next* and *prev* are null.
- *public DoubleNode<T> getNext* (). Returns the value of *next*.
- public DoubleNode<T> getPrev *().* Returns the value of *prev.*
- *public T getData ().* Returns the value of *data*.
- *public void set Next (DoubleNode<T> nextNode*). Stores *nextNode* in *next*.
- *public void setPrev (DoubleNode<T> prevNode*). Stores *prevNode* in *prev*.
- *public void setData (T newData).* Stores *newData* in *data*.

3.3 Class *DoubleList*

This class represents a doubly linked list of nodes of the class *DoubleNode*. The header of this class must be this:

public class DoubleList<T>

This class will have the following instance variables:

- *private DoubleNode<T> head*. This is a reference to the first node in the list.
- *private DoubleNode<T> rear.* This is a reference to the last node in the list.
- *private int numDataItems.* This is the number of nodes in the list.

In this class you must implement the following methods:

- *public DoubleList().* This creates an empty list with zero nodes.
- *public void addData (int index, T newData) throws InvalidPositionException.* This method must add a new node to the list storing *newData*. The node must be inserted in the *index* position of the list. Therefore, if *index* = 0 then the new node must be inserted at the beginning of the list. If *index* = 1, the new node is added after the first node in the list; if *index* = 2, then the new node is added after the second node of the list, and so on. If *index* = *numDataItems*, then the new node is added to the end of the list. If *index* < 0 or *index* > *numDataItems* an *InvalidPositionException* must be thrown.

- *public DoubleNode<T> getNode(int index) throws InvalidPositionException.* Returns the node that is at the *index* position of the list. If *index* = 0, then the first node of the list must be returned; if *index* = 1 then the second node of the list is returned, and so on. If *index* = *numDataItems* - 1 then the last node of the list is returned. If *index* < 0 or *index* ≥ *numDataItems* then an *InvalidPositionException* must be thrown.
- *public void removeData(int index) throws InvalidPositionException*. Removes the node that is at the *index* position of the list (see method *getNode* to see how indices are determined). If *index* < 0 or *index* ≥ *numDataItems* then an *InvalidPositionException* must be thrown.
- *public T getData(int index) throws InvalidPositionException.* Returns the data stored in the node located at the *index* position of the list (see description of method *getNode* to see how indices are determined). If *index* < 0 or *index* ≥ *numDataItems* then an *InvalidPositionException* must be thrown.
- *public void setData(int index, T newData) throws InvalidPositionException.* Store *newData* at the node in position *index* of the list (see description of method *getNode* to see how indices are determined). If *index* < 0 or *index* ≥ *numDataItems* then an *InvalidPositionException* must be thrown.

3.4 Class *BoardGameLinked*

This class represents the board game where the snake moves around eating apples. This class will have the following private instance variables:

- *int boardLength:* the number of columns of the grid on the game board.
- *int boardWidth*: the number of rows of the grid.
- *SnakeLinked theSnake:* an object of the class *SnakeLinked* representing the snake.
- *DoubleList<String>[] board:* an array of doubly linked lists. The first entry of the array is a reference to a linked list representing all the tiles in the first row of the game board; the second entry of the array is a reference to a linked list representing all the tile in the second row of the game board, and so on.
  Each node of the doubly linked list referenced by *board*[*i*] represents one tile of the corresponding row of the game board. Each node of the doubly linked list stores one of the following strings:
  - "empty": if the corresponding tile of the grid is empty.
  - "apple": if the corresponding tile of grid contains an apple.
  - "scissors": if the corresponding tile of the grid contains a pair of scissors.
  - "rock": if the corresponding tile of the grid contains a snake-killing rock.

For example, for the following game board the corresponding *board* data structure is shown on the right.

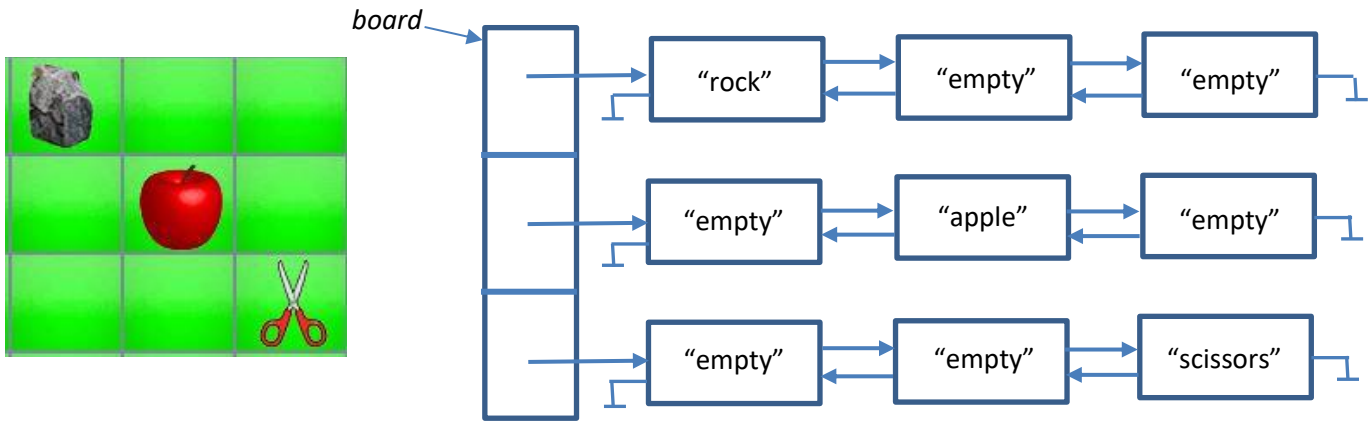

Figure 2. Data structure to represent the game board.

In this class you need to implement the following public methods:

- *public BoardGame(String boardFile):* this is the same method as in Assignment 1, except that now the board game is stored in an object of the class *BoardGameLinked* and the snake is stored in an object of class *SnakeLinked*.
- *public String getObject(int row, int col) throws InvalidPositionException:* returns the string stored in the node with *index = col* of the doubly linked list referenced by *board*[*row*]. For example, for the *board* in Figure 2 invoking *getObject*(1,1) must return "apple" and *getBoard*(0,0) must return "rock". An InvalidPositionException must be thrown if *row* or *col* are negative, if *row* ≥ *boardWidth*, or if *col* ≥ *boardLength*.
- *public void setObject(int row, int col, String newObject) throws InvalidPositionException:* stores *newObject* in the node with *index = col* of the doubly linked list referenced by *board*[*row*]. An InvalidPositionException must be thrown if *row* or *col* are negative, if *row* ≥ *boardWidth*, or if *col* ≥ *boardLength*.
- *public SnakeLinked getSnakeLinked():* returns *theSnake*.
- *public void setSnakeLinked(SnakeLinked newSnake):* stores the value of *newSnake* in instance variable *theSnake*.
- *public int getLength():* returns *boardLength*.
- *public int getWidth():* returns *boardWidth*.

3.5 Class *SnakeLinked*

The class stores the information about the snake as it moves around the board. This class will have two private instance variables:

- *int snakeLength:* this is the number of tiles of the game board occupied by the snake. For example, the snake shown in Figure 1 has a length of 4.
- *DoubleList<Position> snakeBody:* the positions of the tiles of the game board occupied by the snake will be stored in this doubly linked list. The position of the tile with the head of the snake will be stored in the first node of the list (the node referenced by instance variable *head* of class *DoubleList*); the position of the tile where the tail of the snake is, will be stored in last node of the list (the node referenced by *rear*). For example, for the snake in Figure 1, *snakeBody* will contain the following list:

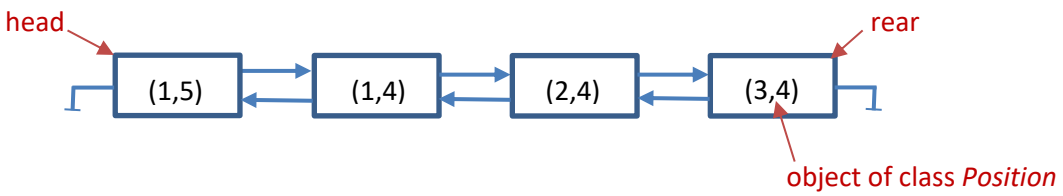

Figure 3. Doubly linked list for the snake in Figure 1.

In this class you need to implement the following public methods.

- *public SnakeLinked(int row, int col):* this is the constructor for the class; the parameters are the coordinates of the head of the snake. Initially the snake has length 1, so in this method the value of the

instance variable *snakeLength* will be set to 1. Instance variable *snakeBody* is initialized so it contains a *DoubleList* of nodes containing *Position* objects. An object of class *Position* will be created storing the values of *row* and *col* and this *Position* object will then be stored in the first node of the list.

- *public int getLength():* returns the value of instance variable *snakeLength*.
- *public Position getPosition(int index):* returns the *Position* object stored in the node of the doubly linked list with the given *index*. So, if *index* = 0 the method returns the *Position* object in the first node of the list; if *index* = 1, it returns the *Position* object in the second node of the list, and so on. This method must return *null* if *index* < 0 or *index* >= *snakeLength*.
- *public boolean snakePosition(Position pos):* returns true if *pos* is in the doubly linked list of *snakeBody*, and it returns false otherwise.
- *public Position newHeadPosition(String direction):* returns the new position that the head of the snake would occupy if the snake moved in the direction specified by the parameter; note that this method does not actually move the snake. The values that *direction* can take are "right", "left", "up" and "down". If, for example, the head of the snake is at (2,3) and *direction* is "right" then the new position would be (2,4); if *direction* is "down" then the new position would be (3,3). If the head is at (0,0) and direction is "up" the new position would be (-1,0).
- *public void moveSnakeLinked(String direction):* moves the snake in the specified direction; this means that the doubly linked list in *snakeBody* must be updated so it contains the positions of the grid tiles that the snake will occupy after it moves in the direction specified by the parameter. For example, for the snake in Figure 1 the doubly linked list in *snakeBody* is shown in Figure 3. If *direction* = "up" then the new doubly liked list in *snakeBody* must be this

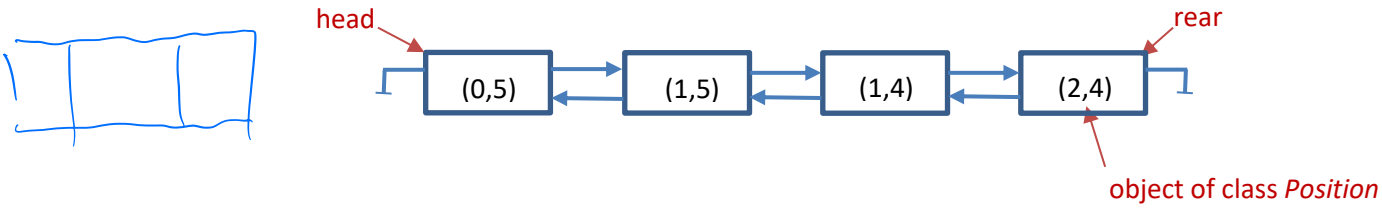

Figure 4. List resulting after moving the snake in Figure 3 up.

- *public void shrink():* decreases the value of *snakeLength* by 1 and deletes the last node in the doubly linked list of *snakeBody*.
- *void grow(String direction):* increases the length of the snake by 1 and moves the snake's head in the direction specified. This method is very similar to method *moveSnake*, except that a new node will be added to the doubly linked list of *snakeBody*. For example, if the snake is as shown in Figure 1, and *direction* = "up" then the new doubly linked list in *snakeBody* would be this

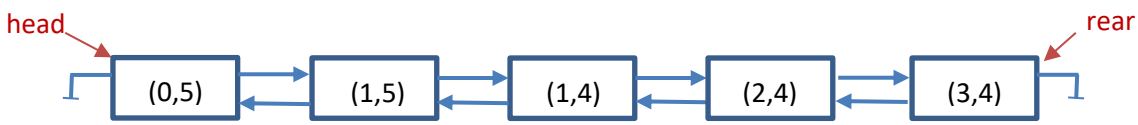

Figure 4. List resulting after moving the snake in Figure 3 up.

In all above classes you can implement more private methods, if you want to, but you cannot implement more public methods. You can also add more private instance variables, but only if they are required. The use of unnecessary instance variables will be penalized.

4. How to Run the Program

Download from the course's webpage the following java classes: PlayChasingGame.java, Chaser.java, MyFileReader.java, and InvalidPositionException and all the image files needed to display the game board. If you are running the program from the terminal place all the files in the same directory and then compile the program by running `javac PlayChasingGame.java` and then run it with `java PlayChasingGame` or with `java PlayChasingGame board_file_name`. If you run the program from Eclipse, read the instructions in the Assignment 1 and Assignment 2.

To start the game press any of the arrow keys. The arrow keys can then be used to change the direction in which the snake moves. Additionally, you can type the following keys:

- `f`: increases the speed of the snake
- `s`: reduces the speed of the snake
- `p`: pauses the game
- `x`: terminates the game
- `d`: prints some debugging information that you might find useful when testing and debugging your program. You must first pause the game to print this information.

Since when running this program, there will be actually several programs running at the same time each controlling a different part of the game, occasionally you might see that some of the tiles of the game board are erased due to timing conflicts between the programs; despite this you should still be able to keep playing the game and the missing tiles will eventually be redrawn.

6. Non-Functional Specifications

- **Assignments are to be done individually and must be your own work**. **Software will be used to detect cheating**.
- You must properly document your code by adding Javadoc comments where appropriate. Add Javadoc comments at the beginning of your classes indicating who the author of the code is and a giving a brief description of the class. Add Javadoc comments to methods to explain what they do and to instance variables to explain their meaning and/or purpose. Also add comments to explain the meaning of potentially confusing parts of your code.
  When deciding where to add comments, you need to use your own judgment. If the meaning of a method, instance variable, or fragment of code is obvious, you do not need to add a comment. If you think that someone else reading a fragment of your code might struggle to understand how the code works, then write a comment. However, try to avoid meaningless comments like these:

```
i = 1;       // initialize the value of i to 1
i = i + 1;   // increase the value of i
if  (i == j) // compare i and j
```

- Use Java coding conventions and good programming techniques:
  - Use meaningful variable and method names. A name should help understand what a variable is used for or what a method does. Avoid the use of variable names without any meaning, like *xxy*, or names, like *flower*, that do not relate to the intended purpose of the variable or method.

- Use consistent conventions for naming variables, methods, and classes. For example, you might decide that names of classes should start with a capital letter, while names of variables and methods should start with a lower-case letter. Names that consist of two or more words like *symbol* and *table* can be combined, for example, using "*camelCasing*" (i.e. the words are concatenated, but the second word starts with a capital letter, for example, *symbolTable*) or they can be combined using underscores as in *symbol_table*. However, you need to be consistent.
- Use consistent notation for naming constants. For example, you can use capital letters to denote constants (`final` instance variables) and constant names composed of several words can be joined by underscores: *TABLE_SIZE*.
- Use constants where appropriate.

- Readability.
  - Use indentation, tabs, and white spaces in a consistent manner to improve the readability of your code. The body of a for loop statement, for example, should have a larger indentation than the statement itself:

```
for (int i = 0; i < TABLE_SIZE; ++i)
        table[i] = 0;
```

  - Positioning of brackets, '{' and '}' to delimit blocks of code should be consistent. For example, if you put an opening bracket at the end of the header of a method:

```
private int method() {
        int position;
```

    then you should not put the bracket in a separate line for another method:

```
private String anotherMethod()
{
        return personName;
```

7. Submitting your Work

You **MUST SUBMIT ALL YOUR JAVA** files through OWL. **DO NOT** put the code inline in the text-box provided by the submission page of OWL. **DO NOT** put a ``package'' line at the top of your java files. **DO NOT** submit a compressed file (.zip, .tar, .gzip, ...); **SUBMIT ONLY** .java files.

Do not submit your .class files. If you do this and do not submit your .java files, your assignment cannot be marked!

8. Marking

What You Will Be Marked On:

- Functional specifications:
  - Does the program behave according to specifications?
  - Does it run with the sample input files provided and produce the correct output?
  - Are your classes implemented properly?
  - Are you using appropriate data structures?
- Non-functional specifications: as described above.
- Assignment has a total of 20 marks.