Assignment One

Nicholas Fiore

Nicholas.Fiore@Marist.edu

September 24, 2022

1 Classes

All the classes created and used in the program (excluding Main)

1.1 Node

Node is the backbone of all the data structures used in this lab. Consider the idea of a metal chain. A single link of a chain is not much by itself. However, as more links are added, the chain grows and now has a purpose. The chain can be as long or as short as necessary, its size dependant on how many links are in the chain.

Node objects are similar to singular links from a chain. By themselves, they don't hold much value, but together can create a powerful data structure. The structure of a Node node itself is quite simple, and contains only two data fields (lines 5-7). The first is a variable to hold the data desired, called ***myData***. This data can be any type desired, *including* other objects or object pointers. In our case, the data stored is a **char** type variable, for the reason that the Nodes and other data structures will be used to check spelling. The other data field is called ***myNext***. This field is how we start creating links in the chain. The ***myNext*** variable holds the address in memory that points to the next Node object in the chain. This allows the creation of a variably sized list by linking Node objects together, one after the other. Because the Node object only holds ***myNext*** and not a variable called say ***myPrev***, it is only possible to traverse a list created with Node "downwards", meaning starting from the first Node in the list and continuing to the last.

The constructors (lines 9-22) of Node are pretty standard. There is an empty/default constructor that initiates a Node object with the default values of ***myData*** (' ') and of ***myNext*** (null). There is a partial constructor that allows you to enter a character as an argument for ***myData*** to be set, and a full constructor that allows you to set both ***myData*** and ***myNext***.

Node requires no specialty methods to function. However, I included a boolean method called ***hasNext()***. This is a simple method that checks to see if the Node object has a value in ***myNext*** that is not **null**, and returns true or false depending on the result. While this method is not necessary (a programmer could just use a `myNode.getMyNext() != null` comparison in their loop), it provides some small functionality and abstraction while being pretty self-documenting with its use.

For added security the data fields are **private** and the class is given accessor and mutator methods for both data fields (lines 24-45).

1.2 Stack

The Stack class is one of the two data structures used in this lab. Stacks operate on a Last-in, first-out basis (LIFO). This means that the most recent element added to the Stack will also be the first object to be removed from the Stack. A common example is with plates or lunch trays. As you add to a stack of lunch trays, you place each new tray on the top. However, if you want to remove one of the trays, you would not lift the entire stack up to grab the bottom tray, you would grab the topmost tray instead (unless it's dirty, then maybe you should grab the next one). A Stack object operates with the same concepts.

Stacks have a single data field, a variable storing a Node pointer address called ***myHead***. This variable keeps track of exactly what Node object in the Stack is currently the top element of the Stack. This marks the top of the Stack as the beginning, and the entire Stack can be iterated through with a loop by starting at ***myHead*** using the ***myNext()*** method from Node. It is also the only way to keep the Stack accessible, as losing ***myHead*** would result in the rest of the elements becoming unreachable.

Like the Node class, Stack contains accessors and mutators to increase the security of the data fields. However, Stack also contains multiple functional methods that allow Stacks to work as stacks.

The first method is ***push()***. This method "pushes" a new element into the Stack, adding it to the top of the Stack as the new head of the Stack. This is done in a very specific order: a Node object that is passed into the method as the ***newNode*** argument is given the Stack's ***myHead*** Node as its ***myNext***. After that, the value of ***myHead*** is then set to be the Node object passed to ***newNode***, making the Node object passed into ***newNode*** become the top of the Stack. The reason that the Stack does not immediately set ***newNode*** as the head is because the other elements of the stack would be lost, with no variable storing the address of the objects.

The next method is ***pop()***. This is the reverse operation of ***push()***. ***pop()*** is a method that removes the top element from the Stack, and returns it in the function. This is simply done by changing the value of ***myHead*** to be the Node object pointed to in the current head's ***myNext***.

The final method is ***isEmpty()***. This is a simple boolean method that checks if the Stack is empty. This is done by checking if the value of `myHead == null`, and returning true of false depending on the value.

1.3 Queue

The Queue is the second main data structure used in this Lab. Queues operate on a first-in, first-out basis. This is comparable to a line of people, or a, well, queue. The customer who is first in line will be served first. Once that customer is served, the next customer will be served, and so on and so forth. New elements added to the Queue are always added to the back, and elements removed are removed from the front.

The two data fields for Queues are as follows: ***myHead*** and ***myTail***. ***myHead*** functions the same was as it does in Stack, keeping track of the element at the front of the queue. ***myTail*** does the same, but for the element at the end of the queue. Technically, ***myTail*** could be omitted, however including it has some benefits. The ***enqueue()*** method (which will be talked about later) adds an item to the end of the Queue. Without a pointer to the element at the end of the Queue, it would have to iterate through the entire list (making it an $O(n)$ operation). By using a little bit extra memory to keep track of ***myTail***, the operation becomes $O(1)$, or constant time.

Queues also have accessors and mutators for added security. The first functional method for a Queue is ***enqueue()***. As stated earlier, this method adds a new element to the end of the Queue. This is done by

setting the ***myNext*** of the current ***myTail*** to be a new Node object passed into the method. This ***newNode*** is then set as the new ***myTail***.

The next method is ***dequeue()***. Like ***pop()*** in Stacks, this removes the element at the front of the Queue by setting the value of ***myHead*** to be the value of ***myNext*** of the current head. This method also returns the element that was dequeued.

The final method is again ***isEmpty()***, and functions exactly as it does in the Stack class.

2 Main Program

This is the main function of the program. The purpose of the program is to read in a .txt with words and/or phrases (one per line), compare the spelling of the word/phrase using Stacks and Queues (ignoring spaces and capitalization), and determines if the word/phrase is a palindrome, and prints it to console if it is.

2.1 Initial Set-up and File Reading

The first section of the program is basic set up for the program. All of the necessary classes are imported before the Main class begins with the main method. A scanner is created to read the input from the keyboard, and a prompt for the user to enter the .txt file to be used is initiated. Variables such as the file being used, the total lines in the .txt, and an array to store each line from the .txt as a String are also created.

The program then enters a try-catch block. It will attempt to execute the rest of the program if successful, but has two catches for exceptions. The first is a file not found exception catch, and the second is a general exception catch.

2.2 Palindromes

If the file is successfully read, the program moves on to the intended functionality. First a scanner is created for the purpose of reading through the .txt. TotLines is set to the amount of lines in the file, and a for loop is used to read the file line by line and then put each String into an array. Once this is complete, a few new variables are created. A Stack and a Queue are created, a counter for the number of palindromes in the file, a flag boolean to be used to check if the word being checked is a palindrome, and a temporary string to be used for String manipulation.

The program then enters a new for loop. This loop will go through every String in the array, which is every line in the .txt file. The flag is set to true initially, and the String being read is put into the temporary string and the spaces are removed while the String is forced to all uppercase letters.

There is then an internal for loop that goes through the current String in ***tempStr***, and both enqueues each letter in the String into the Queue and pushes each letter into the Stack. After this loop, a new for loop is entered. This for loop is the loop that checks if the word is a palindrome. The way it does this is by dequeuing a letter from the Queue and popping a letter from the Stack, and comparing the two characters. If at any point the letters do not match, the flag is set to false. The reason that this works is because of how palindromes are symmetrical, and how Queues and Stacks function. Queues are FIFO, and Stacks are LIFO. Basically, this means with the loop, a Queue will go through the word forwards, while the Stack will go through the word backwards. By the law of symmetry, this would mean every letter would match in each comparison if the word is a palindrome.

The final part of the program is a simple check of the flag. If ***flag*** was every set to false in the previous inner loop, nothing occurs. If the flag was never changed, the current String from the array is printed, and the counter for the total number of palindromes is increased by one. After every String in the array has gone through this process, the outer loop ends and the total number of palindromes is printed.

3 Appendix

3.1 Class source code listings

3.1.1 Main

```
//Utility imports
import java.io.*;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Scanner;

/*
 * The purpose of this program is to take in a file with items, words, phrases, etc. each on separate lines and use various data structures such as linked lists, stacks,
 * and queues, in order to tell if said item/word/phrase is a palindrome (ignoring spaces).
 */

public class Main {
    public static void main(String[] args) {
        /* creates a Scanner object for the input stream (keyboard), prompts the user for the filename/path and stores that into a variable, and then closes the scanner */
        Scanner keyboard = new Scanner(System.in);
        System.out.print("Enter the filename or path: ");
        String fileName = keyboard.nextLine();
        keyboard.close();

        //creates a new file object for the magicitems.txt file or other .txt file
        File file = new File(fileName);
        //long variable to store the total lines of a .txt
        long totLines;
        //array to hold all the items in the list
        String[] itemList;


        /* This try-catch block is for reading in a .txt file, putting each line onto an array, and throwing exceptions if there are any. */
        try {
            //Scanner object for scanning the file
            Scanner fileScanner = new Scanner(file);
            Path path = Paths.get(fileName);

            //grabs the amount of lines within the .txt file and prints it
            totLines = Files.lines(path).count();
            System.out.println("Total lines: " + totLines + "\n");

            //initializes the size of the array with the total lines in the .txt
            itemList = new String[(int)totLines];

            for (int i = 0; i < totLines; i ++) {
                itemList[i] = fileScanner.nextLine();
            }

            //closes the scanner after use to save resources
            fileScanner.close();


            /*
            * This is the main section of the program. It performs all the operations necessary to find the palindromes in a .txt list,
            * including initializing a Stack and Queue object, adding the characters from each word into the Stack and Queue by putting
            * them into Node objects and pushing and enqueuing them, then popping and dequeuing them to compare the letters.
```

```
54              */
55              Queue itemQueue = new Queue();
56              Stack itemStack = new Stack();
57              //counts how many palindrones are in the list
58              int counter = 0;
59              //flag that is dependent on whether or not an item is a palindrome
60              boolean flag;
61              //temporary String object to store the String in the array and remove spaces/convert
                    to uppercase
62              String tempStr = "";
63
64              //for loop goes through every line within the file
65              for (int i = 0; i < totLines; i++) {
66                  //resets flag to true (a word is considered a palindrome until it is proven not)
67                  flag = true;
68                  tempStr = itemList[i].replaceAll("\\s", "").toUpperCase();
69
70                  //pushes each character sequentially into a stack and a queue
71                  for (int j = 0; j < tempStr.length(); j++) {
72                      itemQueue.enqueue(new Node(tempStr.charAt(j)));
73                      itemStack.push(new Node(tempStr.charAt(j)));
74                  }
75
76                  //sequentially pops each Node from the stack and dequeues each node from the queue
                        together. If at any point the character
77                  //in the Node object of the popped object does not match the character in the Node
                        that was dequeued, flag is set to false
78                  //as the word cannot be a palindrome in that case.
79                  for (int j = 0; j < tempStr.length(); j++) {
80                      if (itemQueue.dequeue().getMyChar() != itemStack.pop().getMyChar())
81                          flag = false;
82                  }
83
84                  //if the flag is set to true still, the word is a palindrome and is printed, and
                        the counter is incremented
85                  if (flag) {
86                      System.out.println(itemList[i]);
87                      counter++;
88                  }
89              }
90              System.out.println("\nTotal palindromes: " + counter);
91          } catch(FileNotFoundException ex) {
92                          System.out.println("Failed to find file: " + file.getAbsolutePath());
93          } catch(Exception ex) {
94              System.out.println("Something went wrong.");
95              System.out.println(ex.getMessage());
96          }
97      }
98  }
```

3.1.2 Node

```java
/*
 * The Node class is used to created linked lists of objects in order to more variably control the size of said list without being locked into an array
 */
public class Node {
    /* Data Fields */
    private char myChar = '␣';
    private Node myNext = null;

    /* Constructors */
    //No-arg (default) constructor for creating a default Node object
    public Node() {}

    //Partial constructor for initializing only the Item within the Node
    public Node(char character) {
        myChar = character;
    }

    //Full constuctor for creating a Node object and assigning an character name and next Node object pointer
    public Node(char character, Node next) {
        myChar = character;
        myNext = next;
    }

    /* Accessors and Mutators */
    //Returns the character String of the Node object
    public char getMyChar() {
        return myChar;
    }

    //Returns the pointer for the next Node object in the list
    public Node getMyNext() {
        return myNext;
    }

    //Changes the value of the character String of a Node object to a new String
    public void setMyChar(char myChar) {
        this.myChar = myChar;
    }

    //Changes the pointer of the next pointer of a Node object to a new pointer to a node Object (or NULL)
    public void setMyNext(Node myNext) {
        this.myNext = myNext;
    }

    /* Functions */
    //Checks to see if the Node object has a myNext value != null. Returns boolean depending on result.
    public boolean hasNext() {
        if (this.getMyNext() != null) {
            return true;
        } else {
            return false;
        }
    }

}
```

3.1.3 Stack

```
/*
 * A class to be used for a Stack data structure. Stacks operate on a last in, first out structure. It has 3 operations, including push, pop, and isEmpty.
 * Stacks use the Node class for the elements in the Stack, and always keep track of the element at the top of the Stack
 */

public class Stack {
    /* Data fields */
    private Node myHead = null;

    /* Constructor */
    //empty default constructor
    public Stack() {}

    /* Accessor and Mutator*/
    //Returns the pointer of myHead
    public Node getMyHead() {
        return myHead;
    }

    //Changes the pointer of myHead
    public void setMyHead(Node myHead) {
        this.myHead = myHead;
    }

    /* Functions */
    //adds a new Node object to the Stack
    public void push(Node newNode) {
        newNode.setMyNext(this.getMyHead());
        this.setMyHead(newNode);
    }

    //removes a node object from the top of the stack and returns the node object
    public Node pop() {
        Node val = this.getMyHead();
        this.setMyHead(val.getMyNext());
        return val;
    }

    //checks to see if a stack is empty
    public boolean isEmpty() {
        if (myHead == null) {
            return true;
        } else {
            return false;
        }
    }
}
```

3.1.4 Queue

```
/*
 * A class to be used for a Stack data structure. Stacks operate on a last in, first out structure. It has 3 operations, including push, pop, and isEmpty.
 * Stacks use the Node class for the elements in the Stack, and always keep track of the element at the top of the Stack
 */

public class Stack {
    /* Data fields */
    private Node myHead = null;

    /* Constructor */
    //empty default constructor
    public Stack() {}

    /* Accessor and Mutator*/
    //Returns the pointer of myHead
    public Node getMyHead() {
        return myHead;
    }

    //Changes the pointer of myHead
    public void setMyHead(Node myHead) {
        this.myHead = myHead;
    }

    /* Functions */
    //adds a new Node object to the Stack
    public void push(Node newNode) {
        newNode.setMyNext(this.getMyHead());
        this.setMyHead(newNode);
    }

    //removes a node object from the top of the stack and returns the node object
    public Node pop() {
        Node val = this.getMyHead();
        this.setMyHead(val.getMyNext());
        return val;
    }

    //checks to see if a stack is empty
    public boolean isEmpty() {
        if (myHead == null) {
            return true;
        } else {
            return false;
        }
    }
}
```