Lab One

Joseph McDonough

Joseph.McDonough1@marist.edu

September 18, 2020

1 Nodes

The file Nodes.java is the class file containing the Nodes, Stack, and Queue classes. The nodes are used to form linked lists. The nodes are made up of a string which holds the data, and a Node object which is a reference to the next node in the linked list.

The node constructor can take a string as a parameter, which becomes the data (line 22). The node reference is null by default (line 23).

```
20  public Node(String s)
21  {
22      data = s;
23      reference = null;
24  }
```

The nextNode(Node n) function sets the reference of the node to a specific, preexisting Node, through the parameter(Node n).

```
27  public void nextNode(Node n)
28  {
29      this.reference = n;
30  }
```

2 Stack

2.1 Stack Constructor

Stack utilizes a linked lists and its nodes. The stack constructor is located in the Node.java class file. It takes no parameters and sets the first element of the stack to null as the stack is empty by default.

```
36  public Stack()
37  {
38      stackFirst = null;
39  }
```

To easily keep track of which node is atop the list, the class variable, stackFirst, is a reference to that Node object.

```
33  class Stack {
34      Node stackFirst;
```

2.2 Stack Modifiers

2.2.1 Stack.push

A node added onto a stack is said to be pushed. Therefore, Stack.push puts a node on top of the stack. Stack.push takes a string as a parameter. That string is the data held by the node.

A new node (n) is created using that string (line 53). In order to make sure the stack knows the proper order of nodes, the references have to be correct. A new, temporary node is created called oldTop (line 54). This node represents the previous top of stack node (stackFirst). The new nodes reference gets set to the node that was previously on top (line 55). Upon completion, the top node in the stack is updated to this new node (line 56).

```
51  public void push(String s)
52  {
53      Node n = new Node(s);
54      Node oldTop = stackFirst;
55      n.reference = oldTop;
56      stackFirst = n;
57  }
```

2.2.2 Stack.pop

A node removed from a stack is said to be popped. As stacks are a Last-In-First-Out (LIFO) data structure, it is the newest and top element in the stack that is popped first. In the case where the stack is empty, there is no top node to push, as a result, an error message prints letting the user know (line 63-64).

If the stack is greater than zero, a top node does exists and is stored in the Stack class under stackFirst. As a result, the node to be popped is set to stackFirst and returned (line 66). Prior to being return though, the reference to this node needs to be removed from the stack so it cannot be called again. The node atop the stack points to the node below it (line 67). Upon leaving the stack, the class variable stackFirst adopts the reference and what was formerly the second node in the stack is now the first.

```
60  public Node pop()
61  {
62      Node returnValue = null;
63      if (this.isEmpty()) {
64          System.out.print("Nothing to pop, stack is empty");
65      } else {
66          returnValue = stackFirst;
67          stackFirst = stackFirst.reference;
68      }
69      return returnValue;
70  }
```

3 Queue

3.1 Queue Constructor

Queue utilizes a linked lists and its nodes. The queue constructor is located in the Node.java class file. It takes no parameters and sets the first and last node of the queue to null as the queue is empty by default.

```
77  public Queue()
78  {
79      queueFirst = null;
80      queueLast = null;
81  }
```

To easily keep track of which node is at the top and bottom of the queue, the class variables, queueFirst and queueLast, are references to those Node object.

```
73  class Queue {
74      Node queueFirst;
75      Node queueLast;
```

3.2 Queue Modifiers

3.2.1 Queue.enqueue

A node added onto a queue is said to be enqueued. Therefore, Queue.enqueue puts a node on the back of a queue. Queue.enqueue takes a string as a parameter. That string is the data held by the node. A new node (last) is created using that string (line 95). Its reference is set to null (line 96).

In order to make sure the queue knows the proper order of nodes, the references have to be correct. A new, temporary node is created called secondToLast (line 97). This node represents the previous last node of queue. If the queue is empty, this one node is set to be the first and last node of the queue (line 98-99).

In the event the queue has a size greater than 0, the reference of the secondToLast node is set to the new node (line 101). Upon completion, the class variable queueLast is set to the newly created node (line 102) such that the queue knows which element is last without having to go through its entirety.

```
93  public void enqueue(String s)
94  {
95      Node last = new Node(s);
96      last.reference = null;
97      Node secondToLast = queueLast;
98      if(isEmpty())
99          queueFirst = last;
100     else
101         secondToLast.reference = last;
102     queueLast = last;
103 }
```

3.2.2 Queue.dequeue

A node removed from a queue is said to be dequeued. As queues are a First-In-First-Out (FIFO) data structure, it is the oldest and front element that is released first. In the case where the queue is empty, there is no front node to dequeue, as a result, an error message prints letting the user know (line 110).

If the stack is greater than zero, a front node does exists and is stored in the Queue class under queueFirst. As a result, the node to be dequeued is set to queueFirst and returned (line 113). Prior to being return

though, the reference to this node needs to be removed from the queue so it cannot be called again. Upon leaving the queue, the class variable queueFirst adopts the reference and what was formerly the second node in the queue is now the first (line 114).

```
106  public Node dequeue()
107  {
108      Node returnValue = null;
109      if (this.isEmpty())
110          System.out.print("Nothing to dequeue, queue is empty");
111      else
112      {
113          returnValue = queueFirst;
114          queueFirst = queueFirst.reference;
115      }
116      return returnValue;
117  }
```

4 Main Program

The main program is encased in a try-catch block such that if the magicitems.txt file does not exists, the program is not ran and the catch prints an error letting the user know the file is not found.

4.1 Arrays

The main program utilizes two ArrayLists. Both ArrayLists hold strings. The first one, myMagicItems, holds the magic items in the text file. The magic items are stored as they appear in text file. No case or punctuation changes made. The string normalization process happens later.

The second list, myPalindromes, holds the palindromes as they appear in the text file.

```
19  ArrayList<String> myMagicItems = new ArrayList<String>();
20  ArrayList<String> myPalindromes = new ArrayList<String>();
```

4.2 File Reader

The text file, magicitems.txt, is read using a Scanner. It is assumed that the text file is in the same directory as the main function.

```
18  Scanner reader = new Scanner(new File("magicitems.txt"));
```

With the use of a while loop, the scanner goes through the magicitems.txt. The reader reads each line and takes it as a string.

Here the string is not normalized, that is, the way it appears exactly in the text file is the way it is sent to the myMagicItems ArrayList (line 25) so it can be properly printed later if it is a palindrome.

```
23  while (reader.hasNextLine())
24  {
25      myMagicItems.add(reader.nextLine());
26  }
```

4.2.1 Normalizing Input

In order for palindromes to be detected, the strings need to be normalized in such a way that capitalization and errand punctuation could be ignored. This helper function takes the item string as a parameter. The string is then put into all lower case and all white space, both before, in, and at the end is removed. Then all punctuation and special characters are removed such that only letters remain (line 65). Upon completion, the string is considered normalized and returned so the palindrome evaluator can properly do its job.

```
63  public static String normalizeItems(String item)
64  {
65      return item.toLowerCase().trim().replaceAll(" ","").replaceAll("\\p{Punct}", "");
66  }
```

4.3 Palindrome Detector

Once all the magic items are put into the ArrayLists, its time to go through the list looking for palindromes. Each item in the list has its own stack (line 32) and queue (line 33). The stack is named itemStack and the queue itemQueue. The outer for-loop (line 29) goes through the myMagicItems list while the inner for-loop (line 37) goes through each letter in the current word. It is on line 34 where the string is normalized. The variable normalItem holds the item in its normalized form and that is the string that is later passed into the evalPalindrome function to be used. Although that is the string used, the item as it appears in the text file is what gets added to the palindrome list in the event it is.

```
29  for(int i = 0; i < myMagicItems.size(); i++)
30  {
31
32      Stack itemStack = new Stack();
33      Queue itemQueue = new Queue();
34      String normalItem = normalizeItems(myMagicItems.get(i));
35
36
37      for(int j = 0; j < normalItem.length(); j++)
38      {
39          String letter = Character.toString(normalItem.charAt(j));
40          itemStack.push(letter);
41          itemQueue.enqueue(letter);
42      }
43
44
45      if(evalPalindrome(itemStack, itemQueue, normalItem))
46          myPalindromes.add(myMagicItems.get(i));
47  }
```

4.3.1 Stack Push and Queue Enqueue

Each letter gets put onto the stack and queue at the same time. Once the letters are pushed (line 40) and enqueued (line 41) onto the stack and queue respectively, the helper function can evaluate the stack and queue to see if the word is indeed a palindrome.

```
40  itemStack.push(letter);
41  itemQueue.enqueue(letter);
```

4.3.2 Evaluating For Palindromes

The evalPalindrome function takes the stack, queue, and the normalized word as parameters. The stack pops the letters (line 75) while the queue dequeues the letters (line 76) one by one. Because of the way the data structures work, the letter popping off the stack first is the last letter in the word and the letter dequeuing off the queue first is the first letter. Such that in the word palindrome, the letter "e" would be the first popped off while the letter "p" would be the first dequeued.

Since a palindrome can be read the same way forwards and backwards, these two letters must be the same. If they are, the next set of letters is read until either the end of the word (line 77), or the two letters do not match (line 79).

In the event the two letters do not match, it is impossible for the word to be a palindrome and the helper function returns false (line 80), meaning the word does not get added to myPalindromes. If the while loop expires and all the letters have matched, there is a palindrome to be had.

```
70  public static boolean evalPalindrome(Stack s, Queue q, String item)
71  {
72      int k = 0;
73      while(k < item.length())
74      {
75          String stackPop = s.pop().data;
76          String queueDequeue = q.dequeue().data;
77          if((stackPop.equals(queueDequeue)))
78              k++;
79          else if(!(stackPop.equals(queueDequeue)))
80              return false;
81      }
82      return true;
83  }
```

A true value is returned and the if statement (line 45) allows the magic item to be added to myPalindromes.

```
45  if(evalPalindrome(itemStack, itemQueue, normalItem))
46      myPalindromes.add(myMagicItems.get(i));
```

4.4 Print Palindromes

Upon completion of the myMagicItems list, the outer-most for-loop expires and the code jumps down to line 50. Here, the for-each loop prints out each element in myPalindrome, which is the list of palindrome words that were present in the magicitems text file.

```
50  for(String palindromes : myPalindromes )
51      System.out.println(palindromes)
```