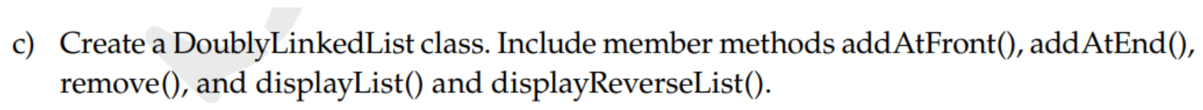
Credit Name: CSE 3020 Computer Science 4

Assignment: DoublyLinkedList

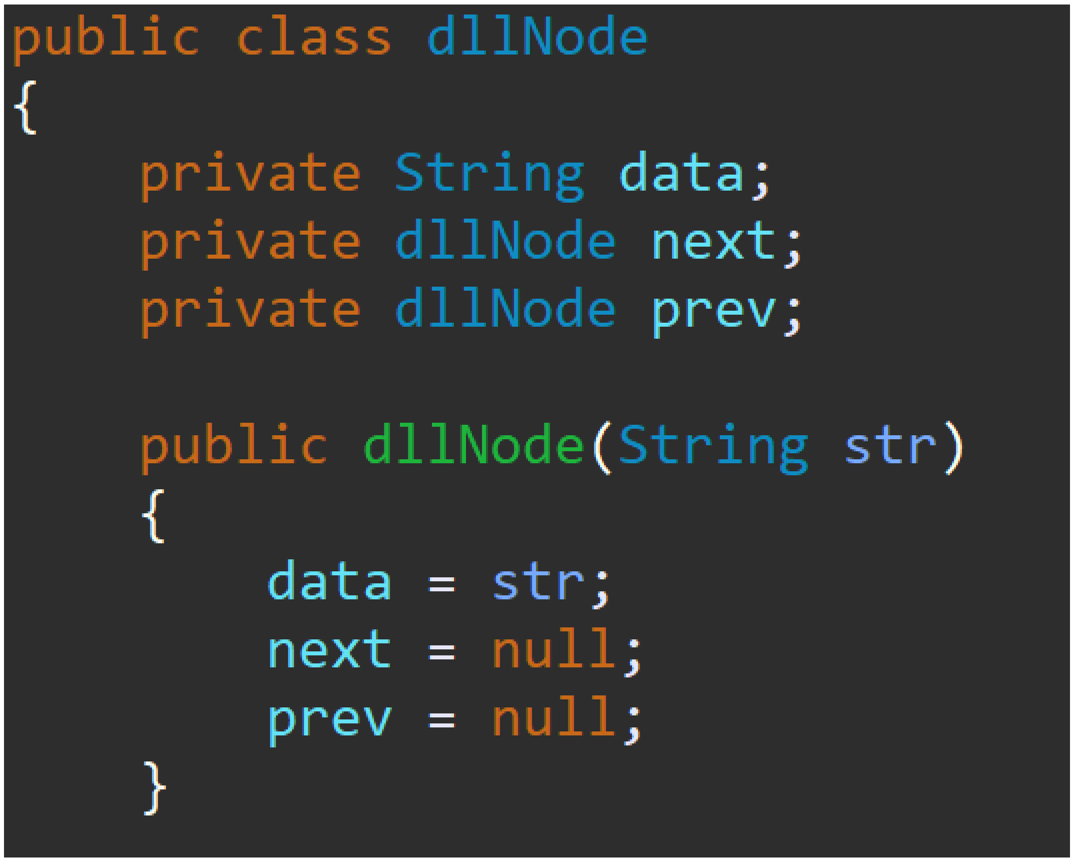
How has your program changed from planning to coding to now? Please Explain

This DoublyLinkedList application has 3 classes: a node class this time that has 2 pointers one to the next node and one to the previous node. Then their is the double linked list which outlines the actual functions of the list and then we have a tester class at the end.

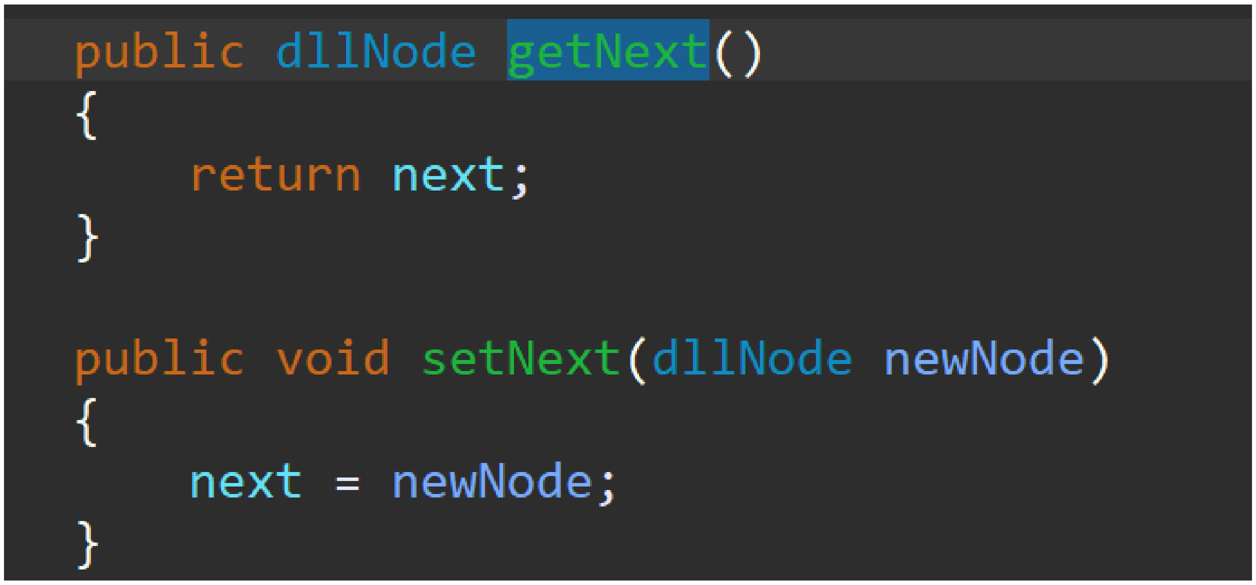

c) Create a DoublyLinkedList class. Include member methods addAtFront(), addAtEnd(), remove(), and displayList() and displayReverseList().

These are the member methods I made based on the prompt of the mastery.

First i made the dllNode Class:

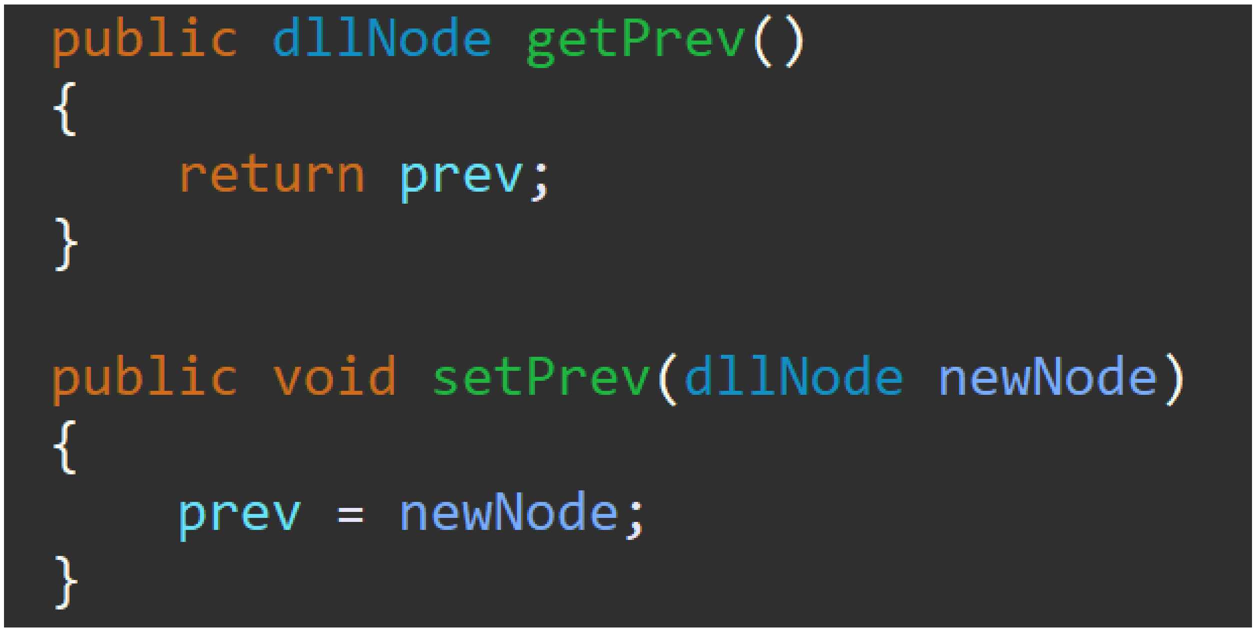

```
public class dllNode
{
    private String data;
    private dllNode next;
    private dllNode prev;

    public dllNode(String str)
    {
        data = str;
        next = null;
        prev = null;
    }
```

I defined it exactly the same way as the previous node class but i added an additional instance variable for the node behind the current one. It was also initialized in the constructor to null because it is not yet related to any other node.



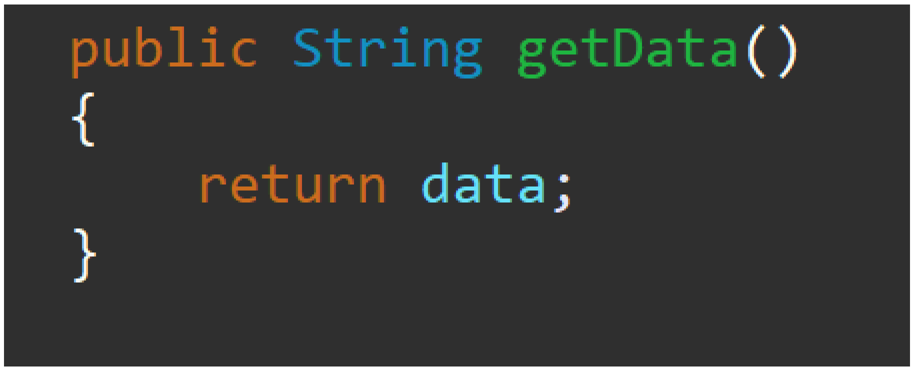
```
public dllNode getNext()
{
    return next;
}

public void setNext(dllNode newNode)
{
    next = newNode;
}
```

There are methods to get the node next to the current one as well as set the next node



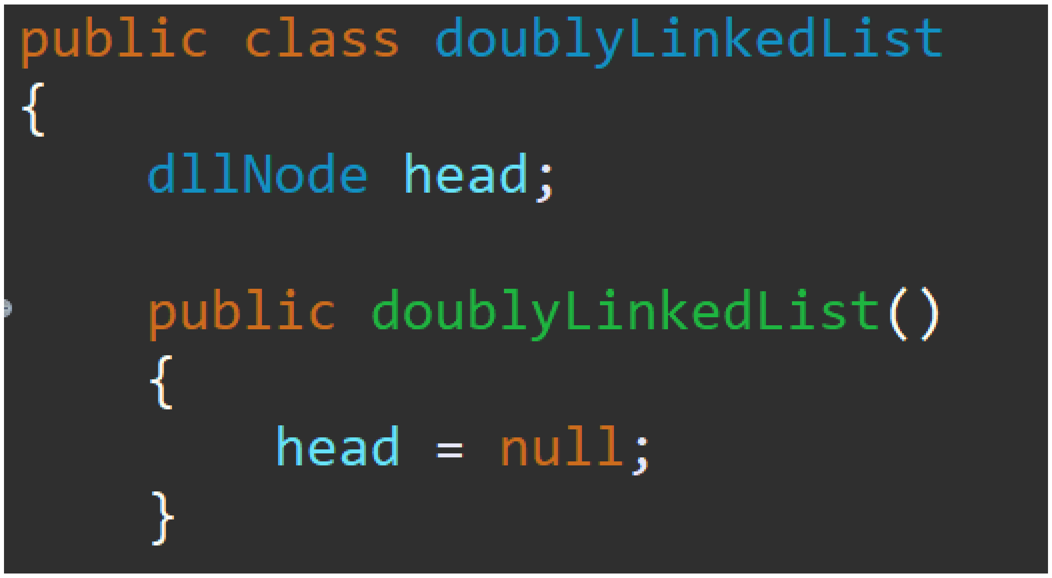
```
public dllNode getPrev()
{
    return prev;
}

public void setPrev(dllNode newNode)
{
    prev = newNode;
}
```

There are methods to get the node behind the current one as well as set the previous node.

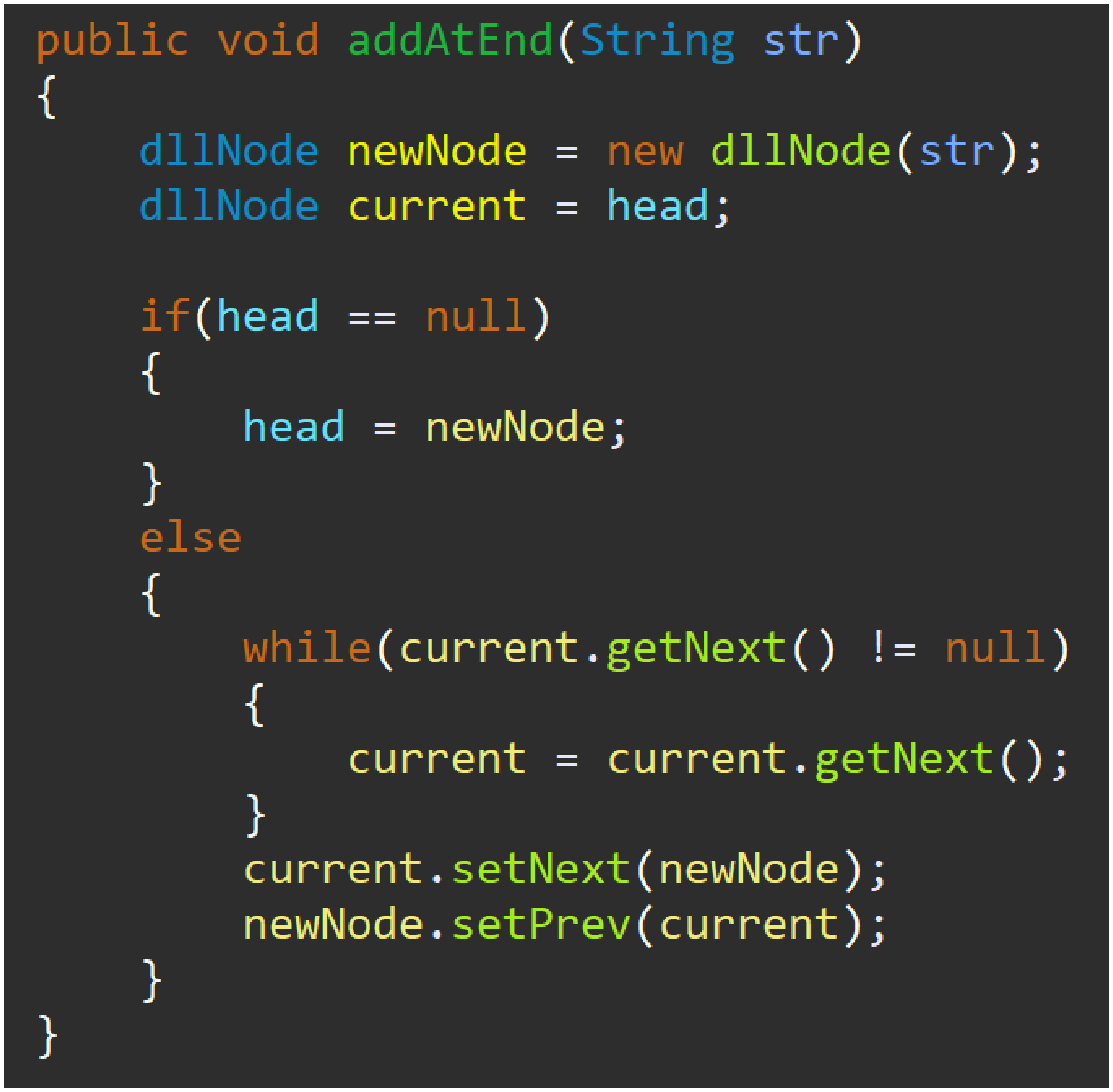

```java
public String getData()
{
    return data;
}
```

Lastly for this class there is a method which returns the data of the node.

The next class defines the doubly linked list with pointers in both directions.

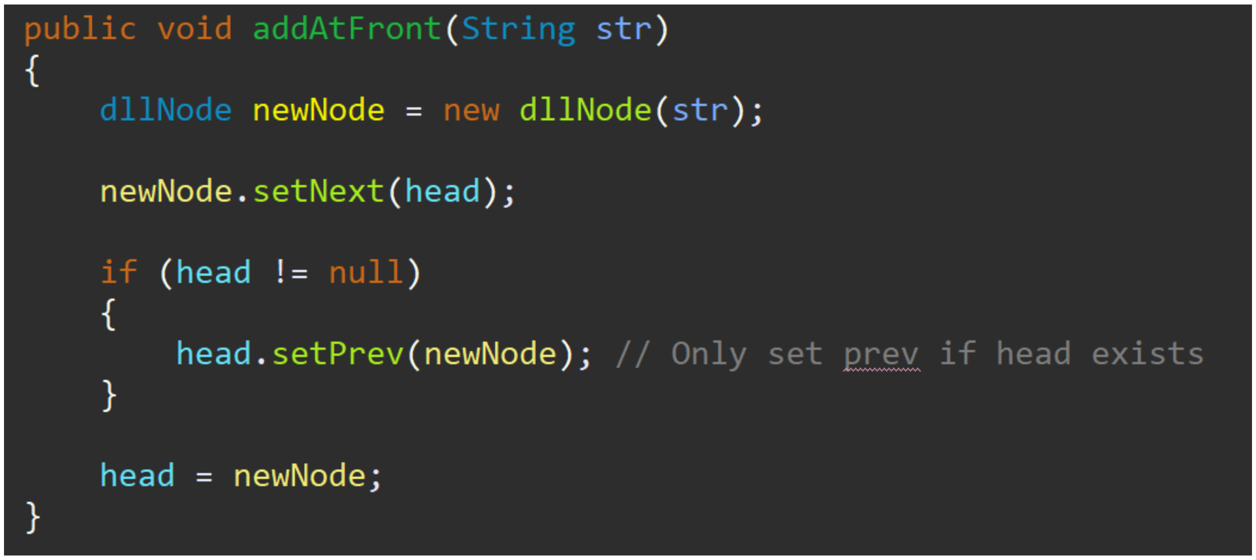

```java
public class doublyLinkedList
{
    dllNode head;

    public doublyLinkedList()
    {
        head = null;
    }
```

Exactly the same way as the singly linked list I defined the head to be an instance variable and initialized it in the constructor to be null.



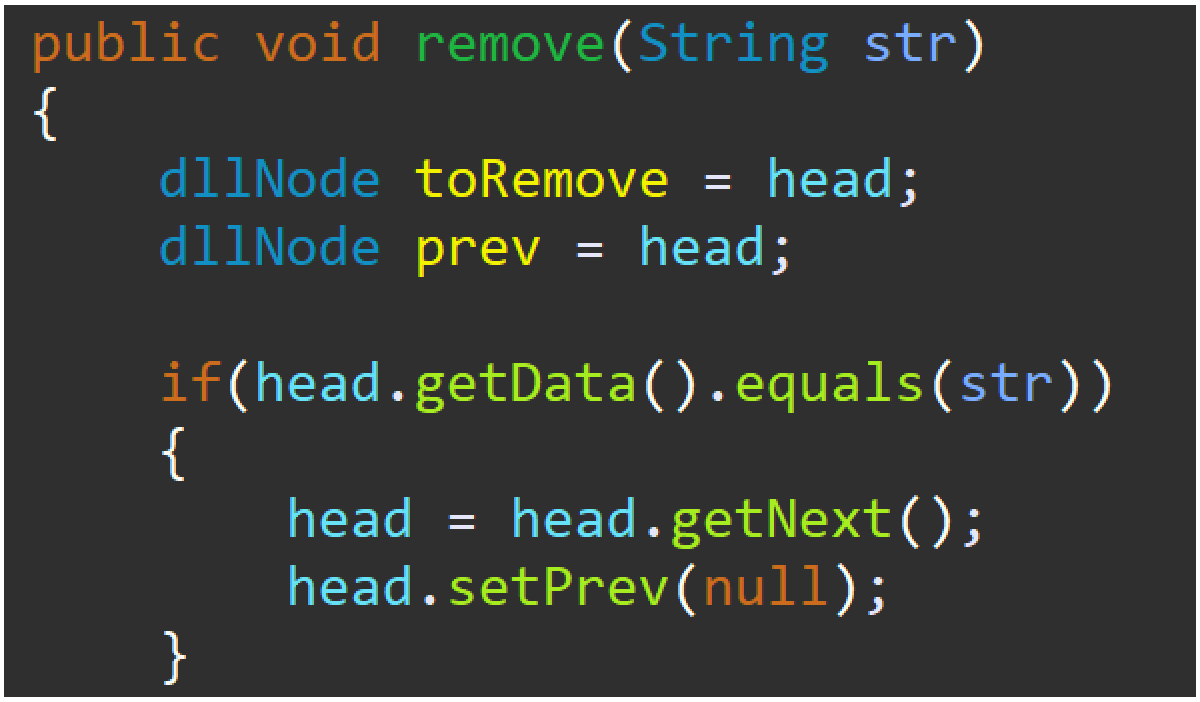
```java
public void addAtEnd(String str)
{
    dllNode newNode = new dllNode(str);
    dllNode current = head;

    if(head == null)
    {
        head = newNode;
    }
    else
    {
        while(current.getNext() != null)
        {
            current = current.getNext();
        }
        current.setNext(newNode);
        newNode.setPrev(current);
    }
}
```

Then I defined a method to add an item at the end of the list. I first declared and initialized the node to be added using the method parameter as the new node's value. Then I declared a variable to be the current position in the list and started it at the head. If the head was null, then the list is empty and the head is the value of the new node. Otherwise while we use a while loop to iterate through the list until we reach the end(indicated by the next value being null). Once we reach the end we set the value next to the current node(at the end of the list) to the new node defined earlier in the method. Since this is a doubly linked list which points both ways i need to set a pointer going backwards from the new node which is pointing to the previous last node.



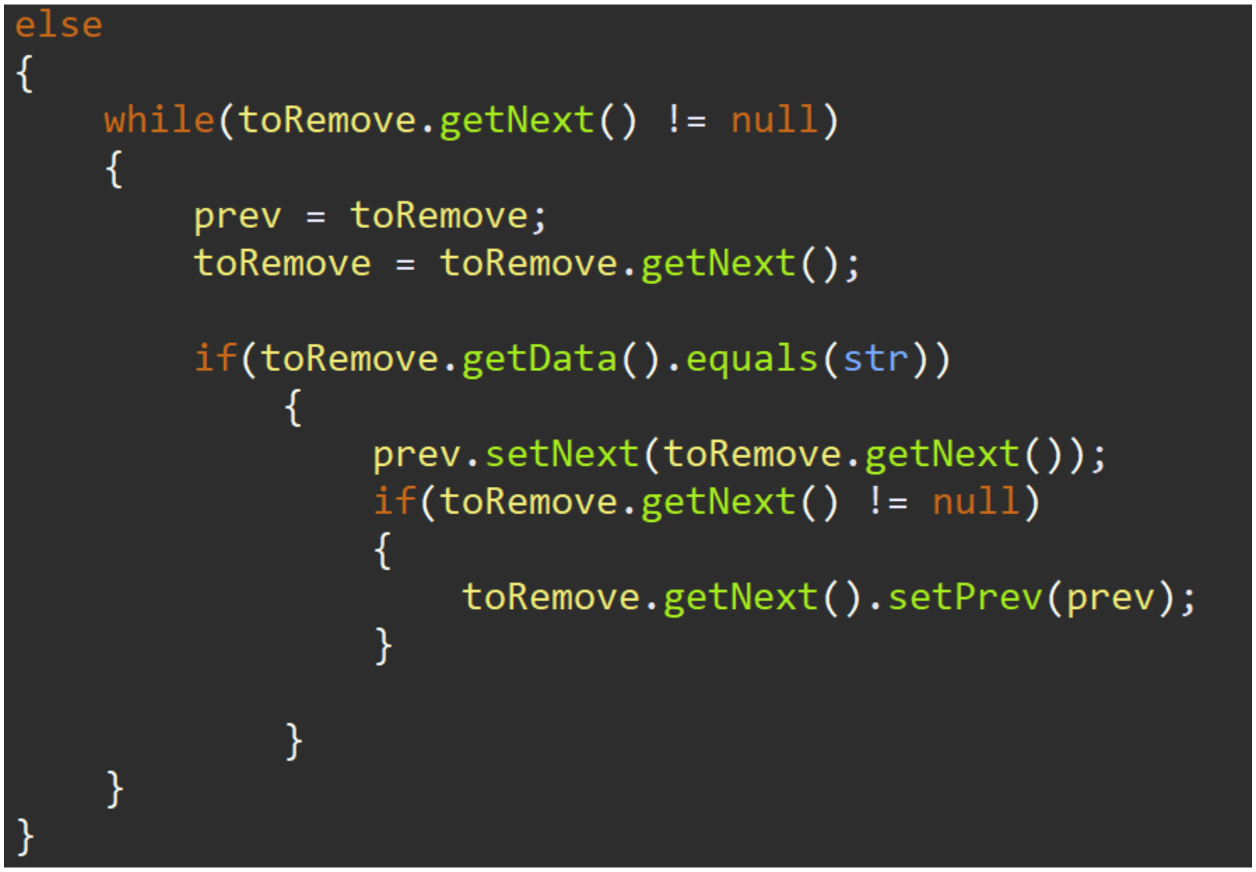
```java
public void addAtFront(String str)
{
    dllNode newNode = new dllNode(str);

    newNode.setNext(head);

    if (head != null)
    {
        head.setPrev(newNode); // Only set prev if head exists
    }

    head = newNode;
}
```

To add a value to the front, we again define a method and intialize the newNode to have the value of the method parameter. Then we set the new nodes' next node to be the head node (so that the 2nd node is now the head node). Then only if the head node isn't null, we set the node behind the head node to be the new node we are adding. Lastly we define the head node to be the new node at the beginning of the list.



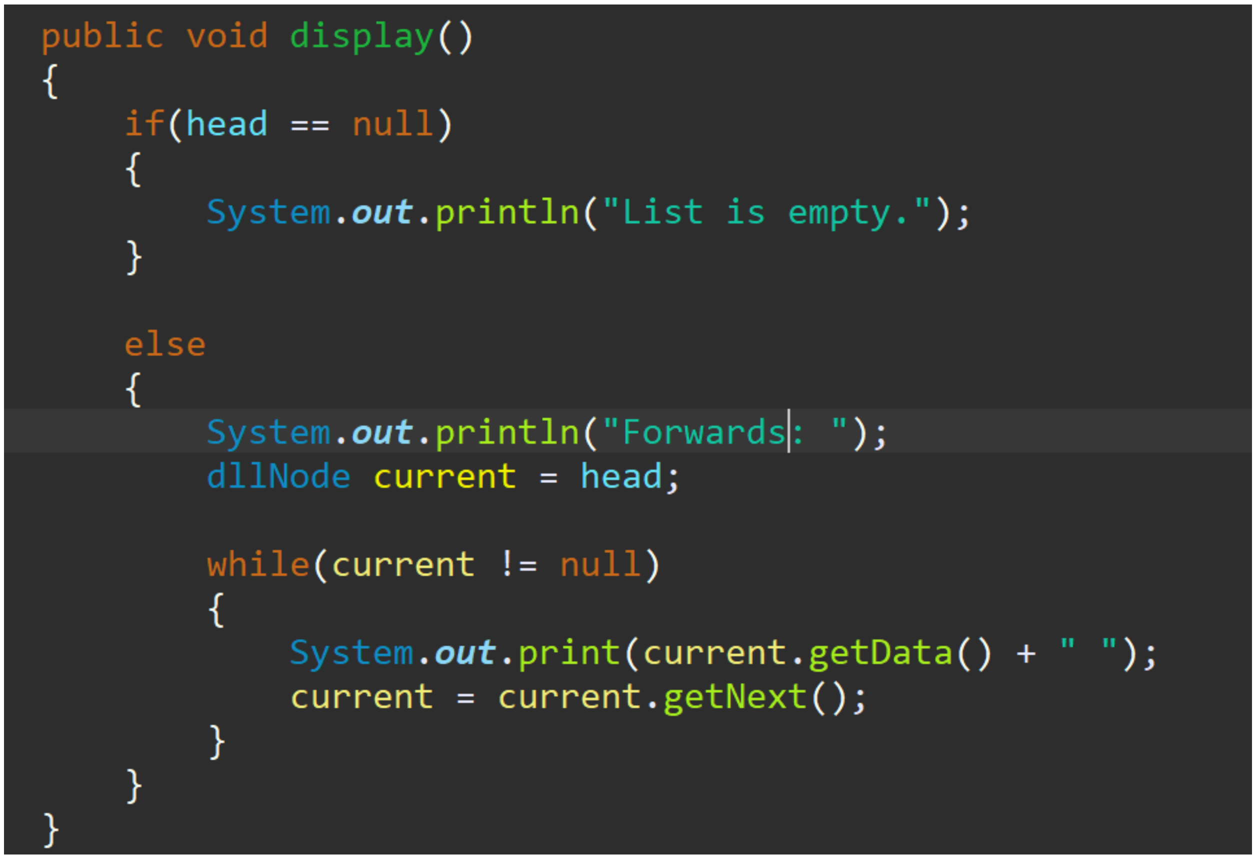
```java
public void remove(String str)
{
    dllNode toRemove = head;
    dllNode prev = head;

    if(head.getData().equals(str))
    {
        head = head.getNext();
        head.setPrev(null);
    }
```

To remove a node from the list, we define 2 variables one which is the node we remove and the node just before that. Both start at head and the idea is that the toRemove variable will always be 1 infront of the prev variable. If the head's data matches the input parameter then we set the head to be the next one across and set the node prior to be null.



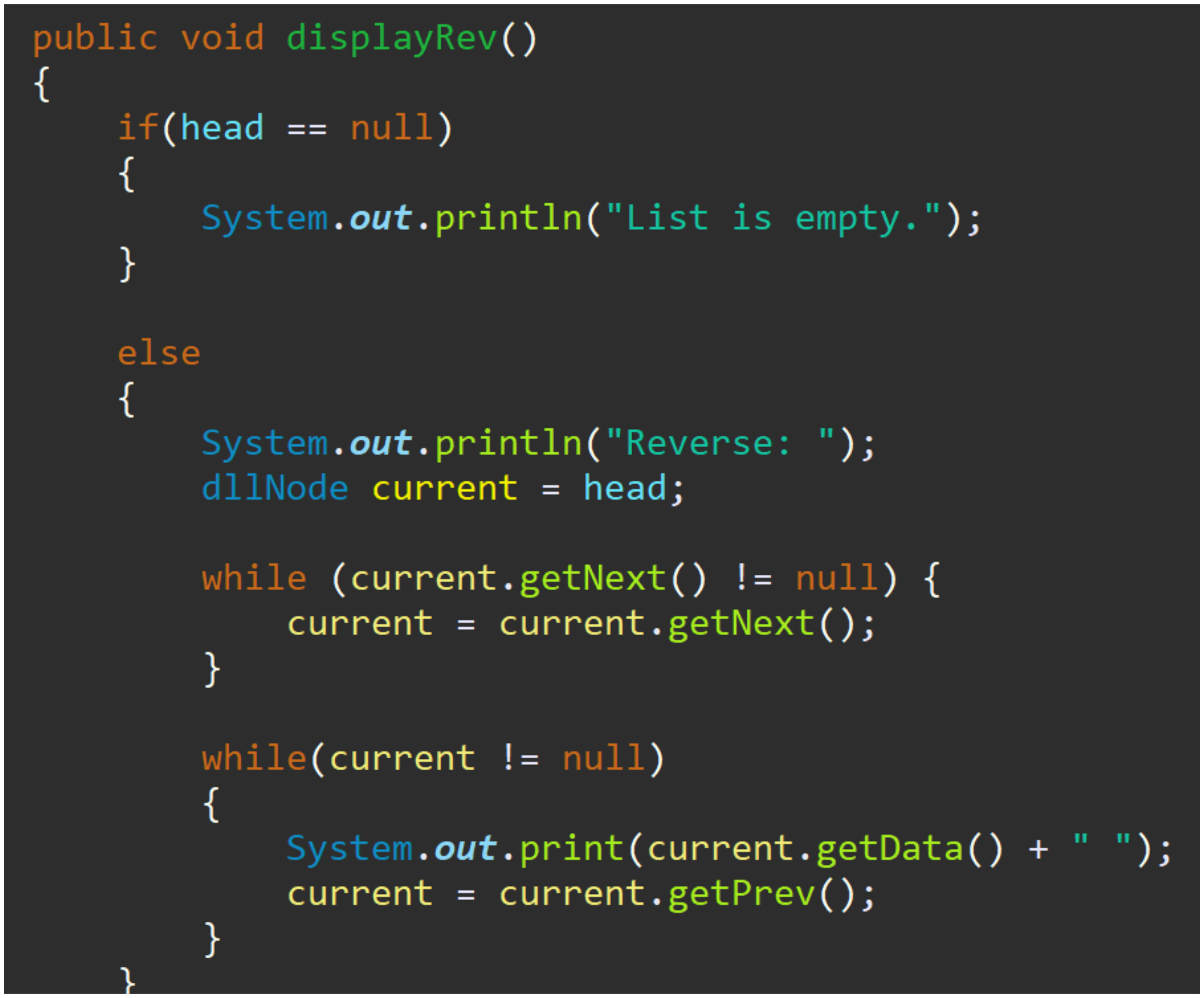
```
else
{
    while(toRemove.getNext() != null)
    {
        prev = toRemove;
        toRemove = toRemove.getNext();

        if(toRemove.getData().equals(str))
            {
                prev.setNext(toRemove.getNext());
                if(toRemove.getNext() != null)
                {
                    toRemove.getNext().setPrev(prev);
                }

            }
    }
}
```

Otherwise we start a while loop which runs while toRemove is not null. In the while loop the prev node is set to the toRemove value and then the toRemove is moved along to the next one. After this the toRemove index is checked to see if it matches the method parameter. If not this while loop will keep going until it does. When a match is found, we set the node just before the one we removed to link to the node AFTER the one we removed (its like we skip past the node we are removing). Then after checking that the node after the one we remove is NOT null then we set up a pointer in the opposite direction to maintain a double link between nodes.



```java
public void display()
{
    if(head == null)
    {
        System.out.println("List is empty.");
    }

    else
    {
        System.out.println("Forwards: ");
        dllNode current = head;

        while(current != null)
        {
            System.out.print(current.getData() + " ");
            current = current.getNext();
        }
    }
}
```

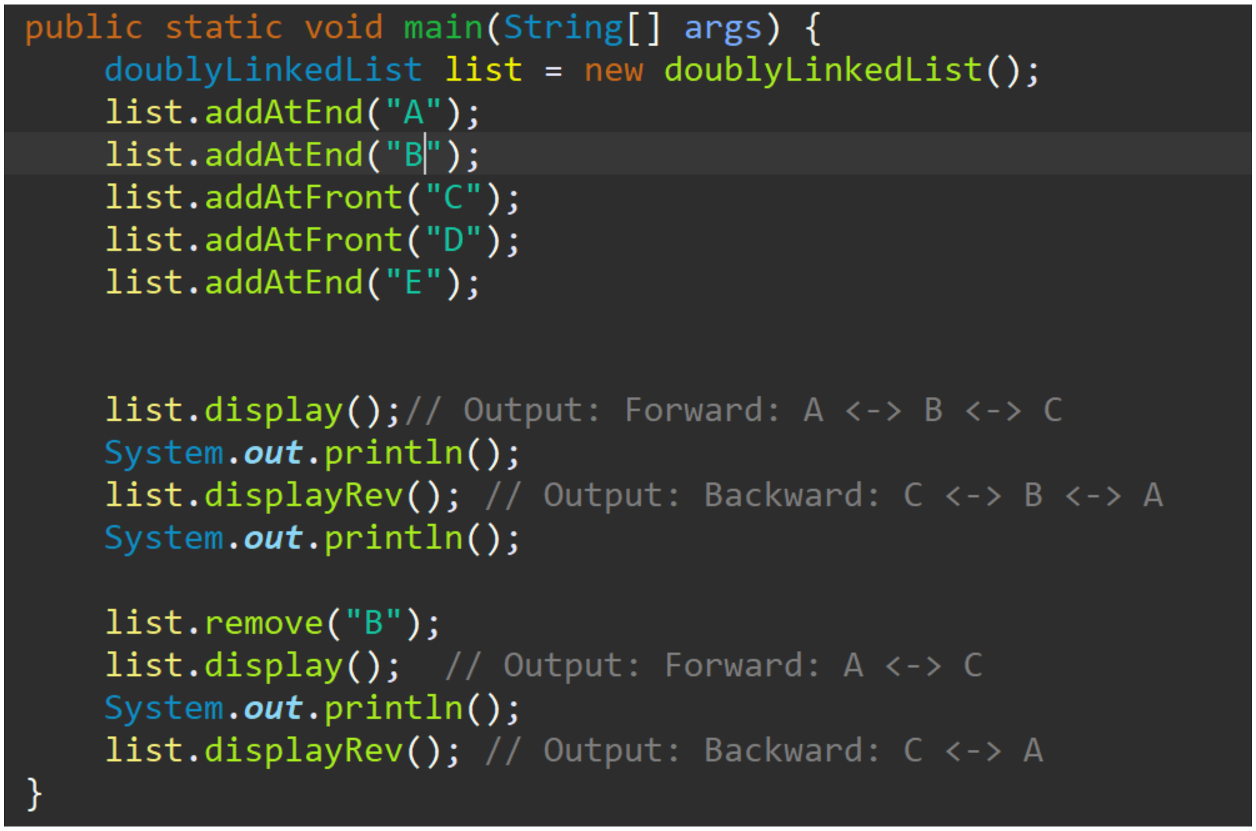
Then to display the list, we first handle the case that the list is empty if the head is null. Otherwise we set up a pointer node labelled current which starts at the head value and we then have a while loop which prints the data from every node in the list and moves the pointer node one across. This is repeated until we reach the end of the list.



```
public void displayRev()
{
    if(head == null)
    {
        System.out.println("List is empty.");
    }

    else
    {
        System.out.println("Reverse: ");
        dllNode current = head;

        while (current.getNext() != null) {
            current = current.getNext();
        }

        while(current != null)
        {
            System.out.print(current.getData() + " ");
            current = current.getPrev();
        }
    }
```

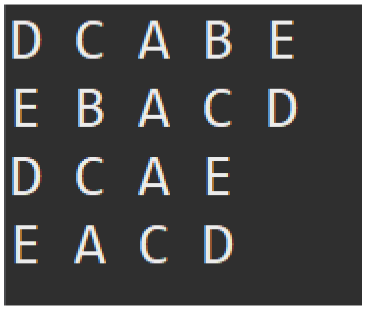
Lastly, To display the list in the reverse order we start by handling the same initial null case. Then the only difference is that we start from head and iterate through until we reach the end. And then we do the same while loop as the forward display but we instead go back towards the initial position while printing each node value.

Our last class is a simple tester class to ensure all methods work as intended.



```java
public static void main(String[] args) {
    doublyLinkedList list = new doublyLinkedList();
    list.addAtEnd("A");
    list.addAtEnd("B");
    list.addAtFront("C");
    list.addAtFront("D");
    list.addAtEnd("E");


    list.display();// Output: Forward: A <-> B <-> C
    System.out.println();
    list.displayRev(); // Output: Backward: C <-> B <-> A
    System.out.println();

    list.remove("B");
    list.display();  // Output: Forward: A <-> C
    System.out.println();
    list.displayRev(); // Output: Backward: C <-> A
}
```

We set up the main method and intialize the doubly linked list. We then add letters, some at the end and some infront. We display them forward and backward. Then we remove a letter and then display the list again. This shows how all the methods work in a simple easy to see way.



```
D C A B E
E B A C D
D C A E
E A C D
```

This is an example of the code above.

a) What advantages does this structure offer? What disadvantages?

b) Compare the doubly-linked list with the singly linked list implemented in this chapter. When would you choose to use one rather than the other?

   a. ***We can travel in both directions making easier operations with both ends of the list. The disadvantage is that the code is more complex and takes up more storage.***

b. ***You would use the doubly linked list when you have more legroom for memory and require bidirectional travel in a list, like making backwards and forwards buttons for a program. On the other hand, when you are more constrained by memory you would want to use a singly linked list.***