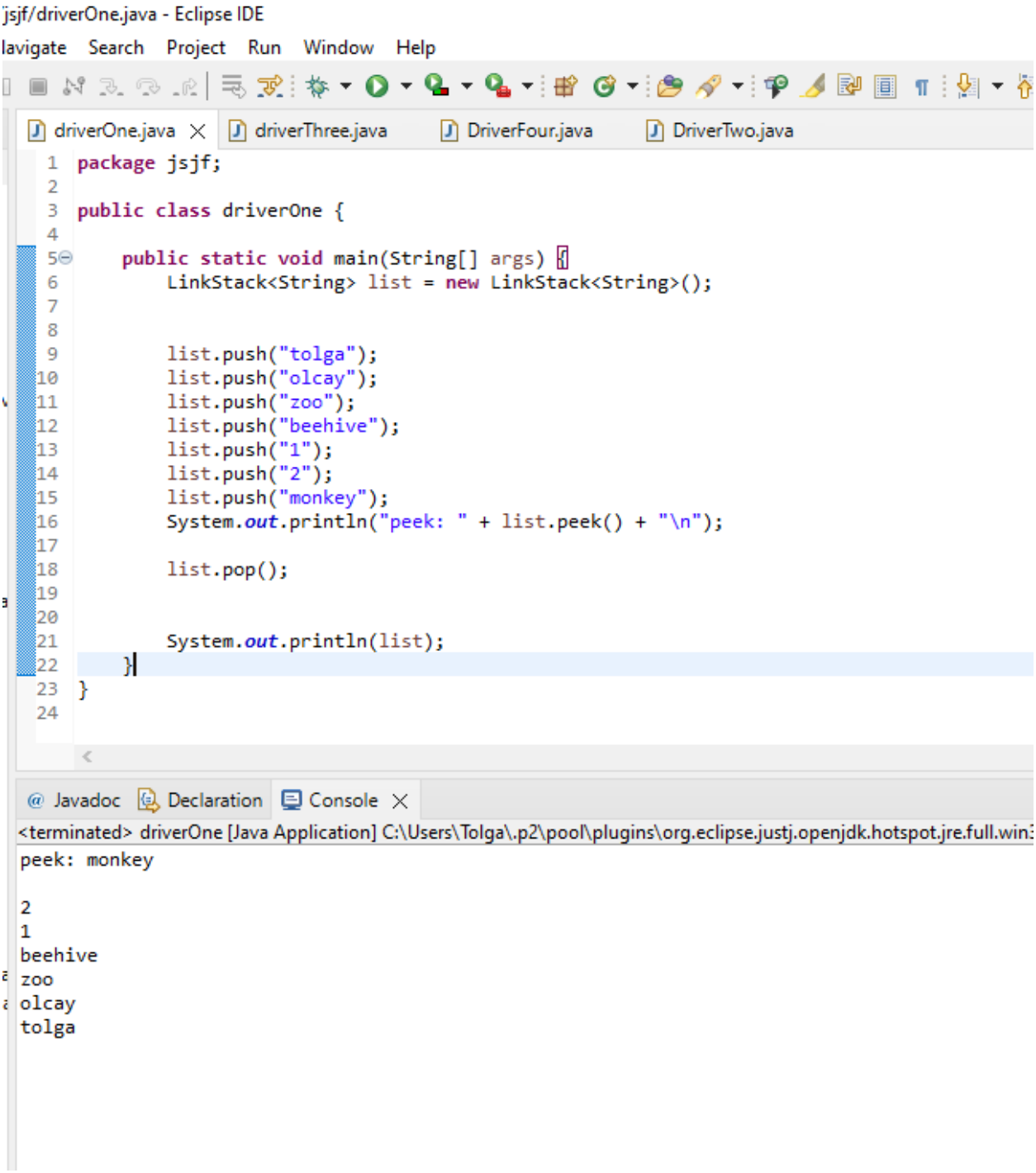
COMP 2231 Assignment 3

T00666715

Tolga Olcay

1. Stack using Linked List

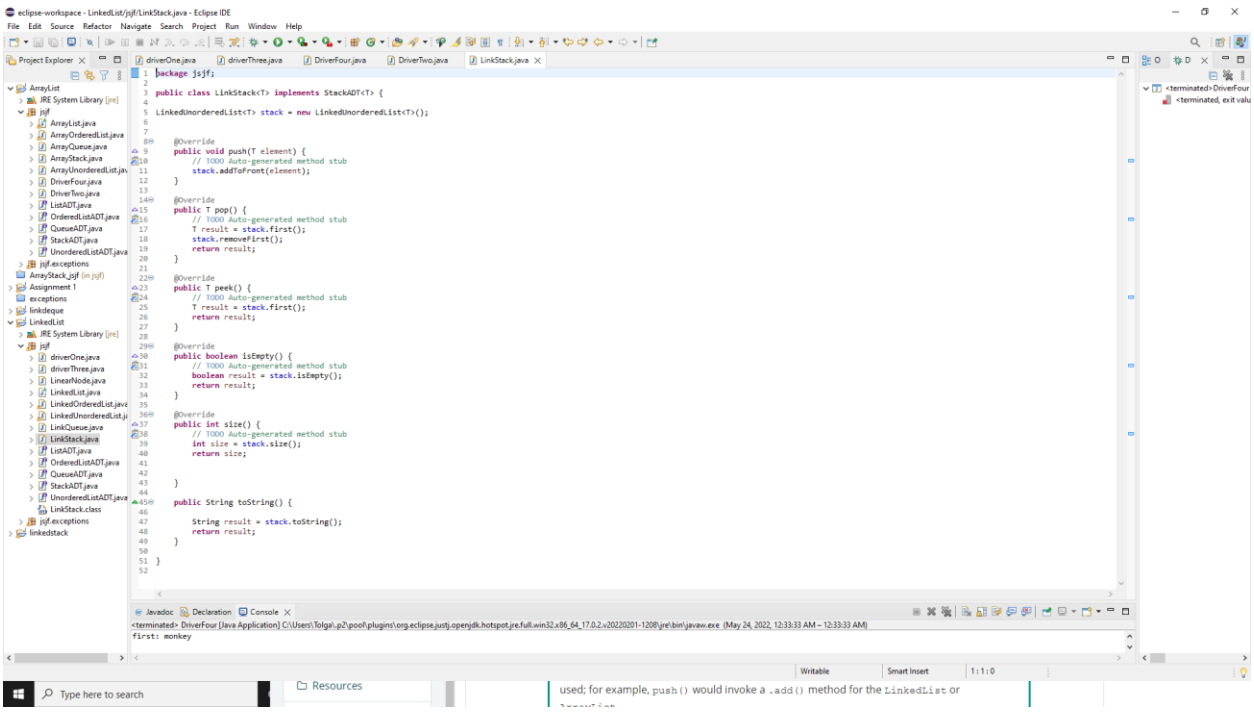



Here is the result of a stack using a linked list. I use the same instructions for all the drivers that use stacks.

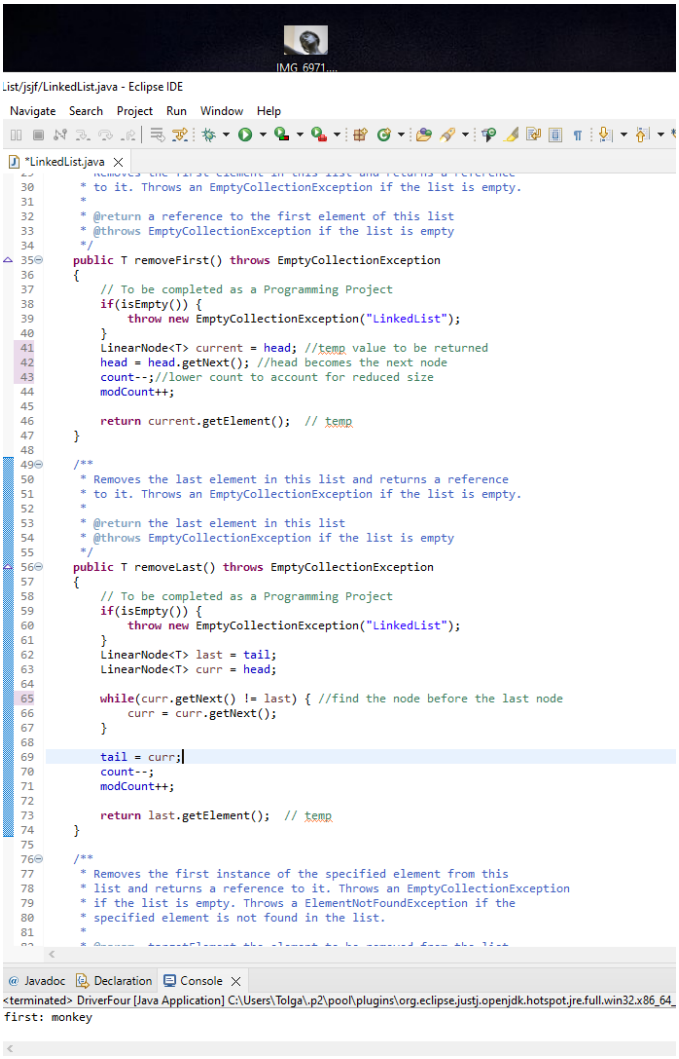
As you can see, when we peek, the string is monkey because that was the last value pushed on the stack.
When we pop and then print the stack, we can notice that monkey is removed from the stack.

Now let me demonstrate how this works:

First Lets look at LinkStack




By using functions from the LinkedList and the LinkedUnorderedList, we can implement the methods of the LinkStack via encapsulation. Lets look at LinkedUnorderedList, LinkedStack, and I **will also show you LinkedOrderedList** (even though it is not necessary)




Here you can see the RemoveFirst() function. It is used in the LinkStack in the pop() method. As you can see I have also completed the removeLast() function even though it is not needed for the LinkStack.

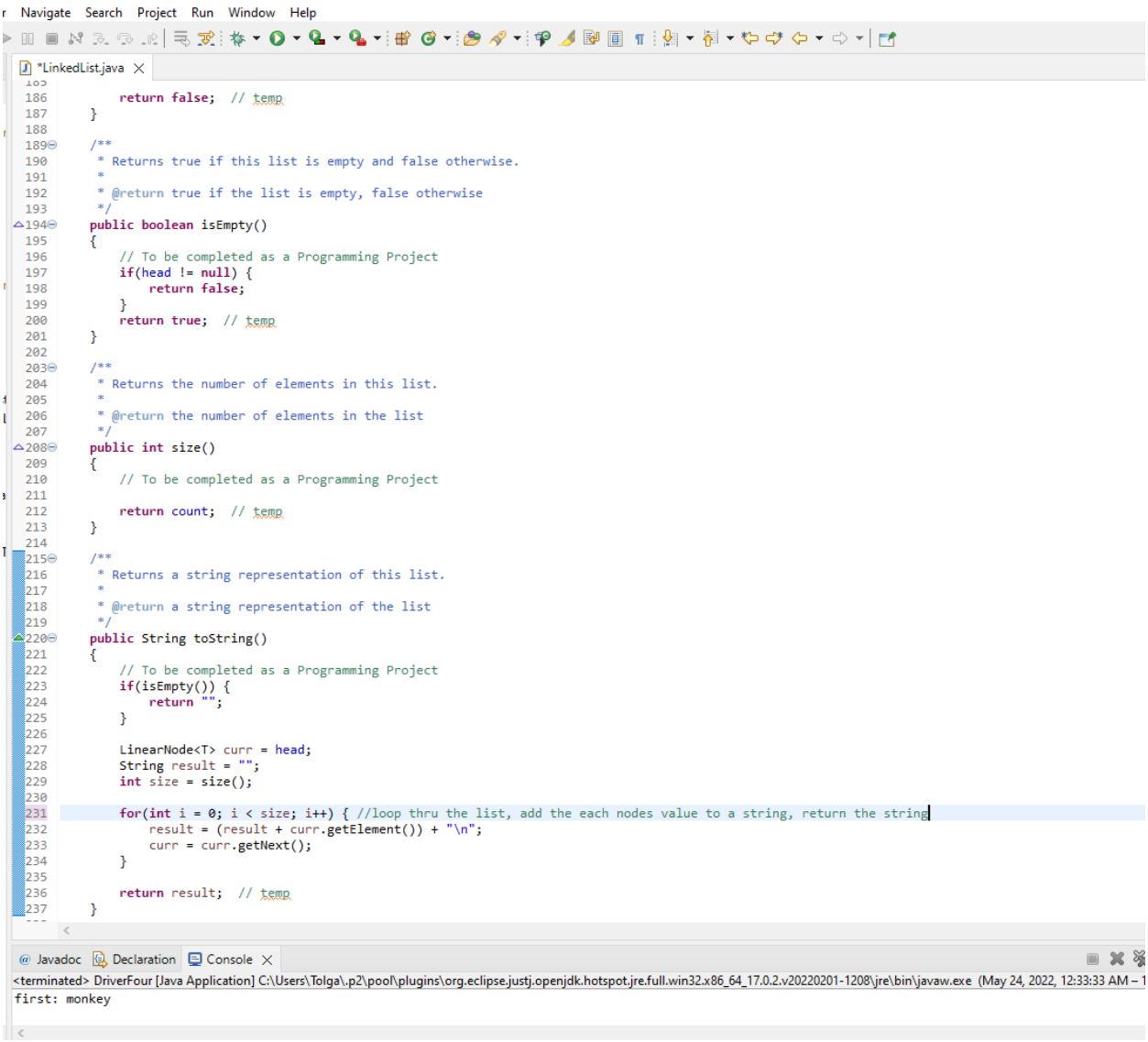
List/jsjf/LinkedList.java - Eclipse IDE

Navigate Search Project Run Window Help

*LinkedList.java

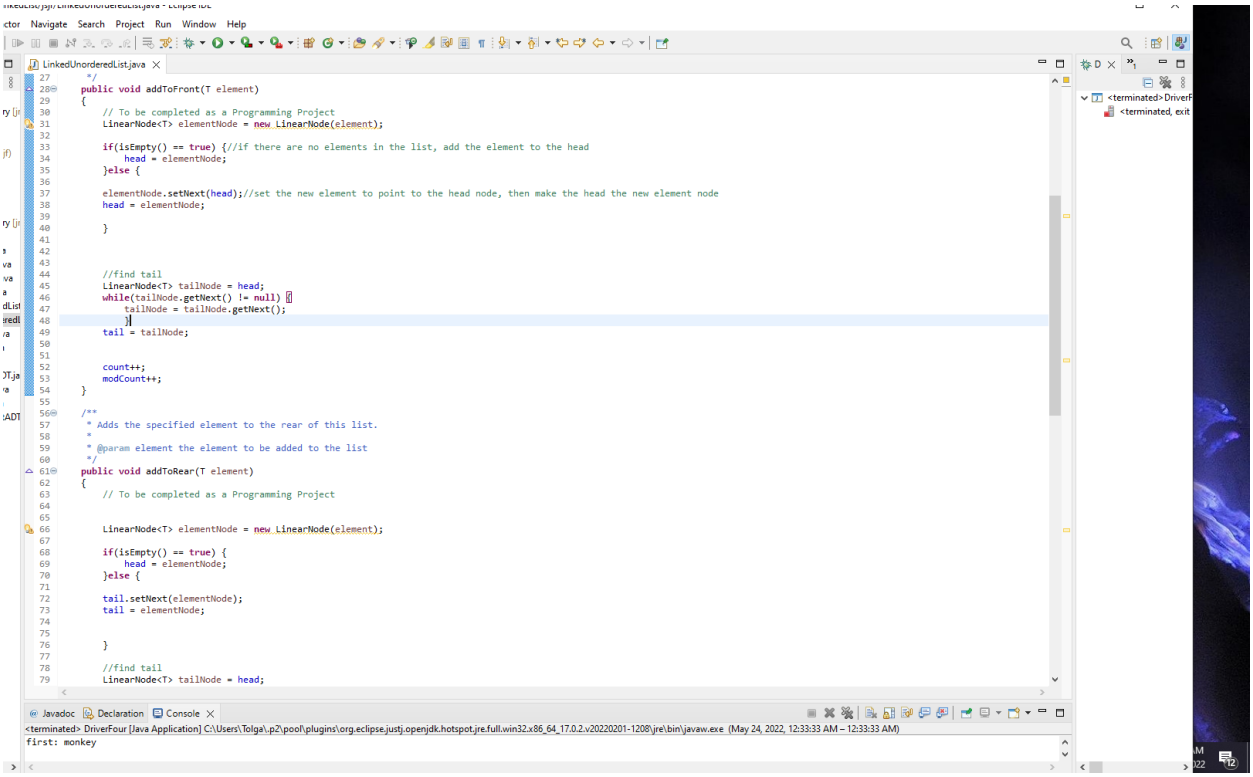
```
133    public T first() throws EmptyCollectionException
134    {
135        // To be completed as a Programming Project
136        if(isEmpty()) {
137            throw new EmptyCollectionException("LinkedList");
138        }
139
140        return head.getElement();  // temp
141    }
142
143    /**
144     * Returns the last element in this list without removing it.
145     *
146     * @return the last element in this list
147     * @throws EmptyCollectionException if the list is empty
148     */
149    public T last() throws EmptyCollectionException
150    {
151        // To be completed as a Programming Project
152        if(isEmpty()) {
153            throw new EmptyCollectionException("LinkedList");
154        }
155
156        return tail.getElement();  // temp
157    }
158
159    /**
160     * Returns true if the specified element is found in this list and
161     * false otherwise. Throws an EmptyCollectionException if the list
162     * is empty.
163     *
164     * @param  targetElement the element that is sought in the list
165     * @return true if the element is found in this list
166     * @throws EmptyCollectionException if the list is empty
167     */
168    public boolean contains(T targetElement) throws EmptyCollectionException
169    {
170        // To be completed as a Programming Project
171        if(isEmpty()) {
172            throw new EmptyCollectionException("LinkedList");
173        }
174
175        int size = size();
176        LinearNode<T> curr = head;
177
178        for(int i = 0; i < size; i++) { //loop thru the list, if the target elmenet is in the list, return true
179            if (targetElement == curr.getElement() ) {
180                return true;
181            }
182            curr = curr.getNext();
183        }
184
185
```

The first() function in the LinkedList class is very useful for the pop() and peek() function in the LinkStack. (I probably should have used the contains() function in the LinkedUnorderedList for the addAfter() function, but I just thought of that now!)




The isEmpty, size, and toString functions in the LinkedList class are all used in the LinkStack class to findout if the stack is empty, find the size, and to print the stack.

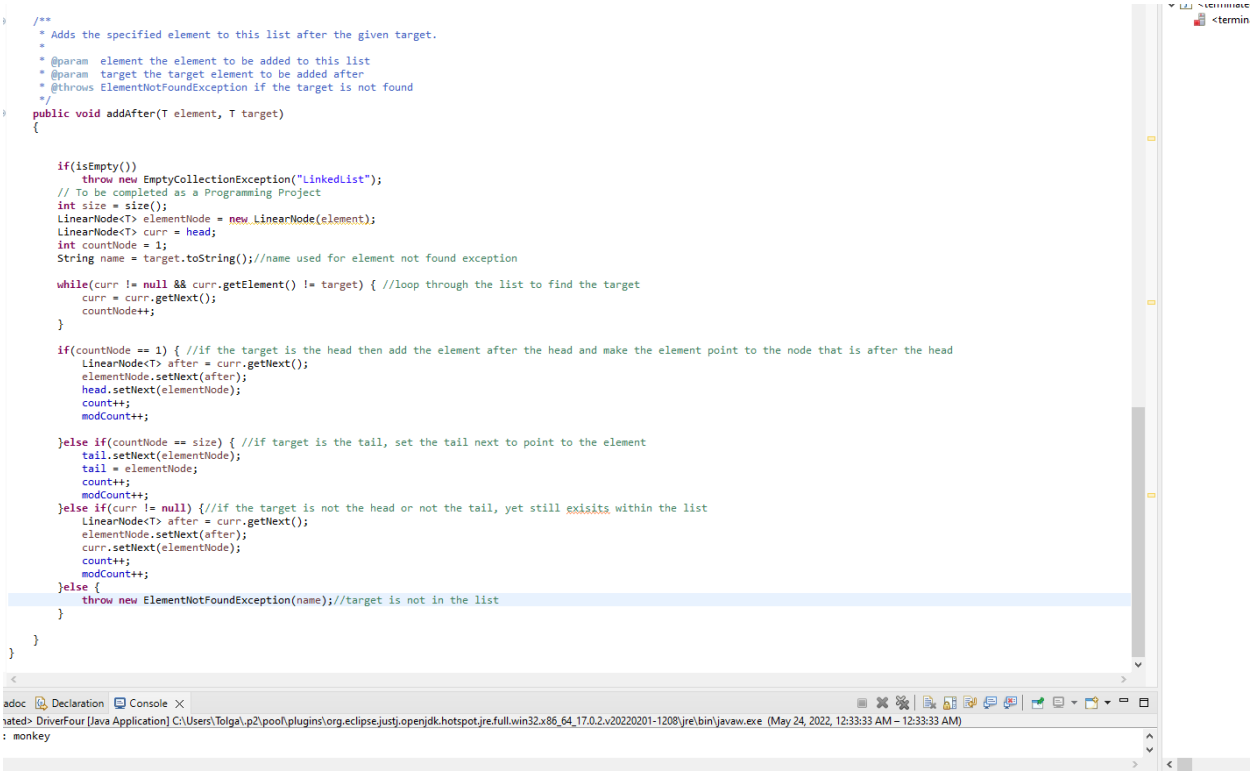
Now lets look at the LinkUnorderedList:



```
    */
    public void addToFront(T element)
    {
        // To be completed as a Programming Project
        LinearNode<T> elementNode = new LinearNode(element);

        if(isEmpty() == true) {//if there are no elements in the list, add the element to the head
            head = elementNode;
        }else {

        elementNode.setNext(head);//set the new element to point to the head node, then make the head the new element node
        head = elementNode;

        }



        //find tail
        LinearNode<T> tailNode = head;
        while(tailNode.getNext() != null) {
            tailNode = tailNode.getNext();
            }
        tail = tailNode;


        count++;
        modCount++;
    }

    /**
     * Adds the specified element to the rear of this list.
     *
     * @param element the element to be added to the list
     */
    public void addToRear(T element)
    {
        // To be completed as a Programming Project


        LinearNode<T> elementNode = new LinearNode(element);

        if(isEmpty() == true) {
            head = elementNode;
        }else {

        tail.setNext(elementNode);
        tail = elementNode;


        }

        //find tail
        LinearNode<T> tailNode = head;
```

```
Javadoc  Declaration  Console
<terminated> DriverFour [Java Application] C:\Users\Tolga\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.2.v20220201-1208\jre\bin\javaw.exe  (May 24, 2022, 12:33:33 AM – 12:33:33 AM)
first: monkey
```

The addToFront and addToRear methods were both required to be completed as part of the programming project, however, ONLY THE addToFront() function was used for the LinkStack.



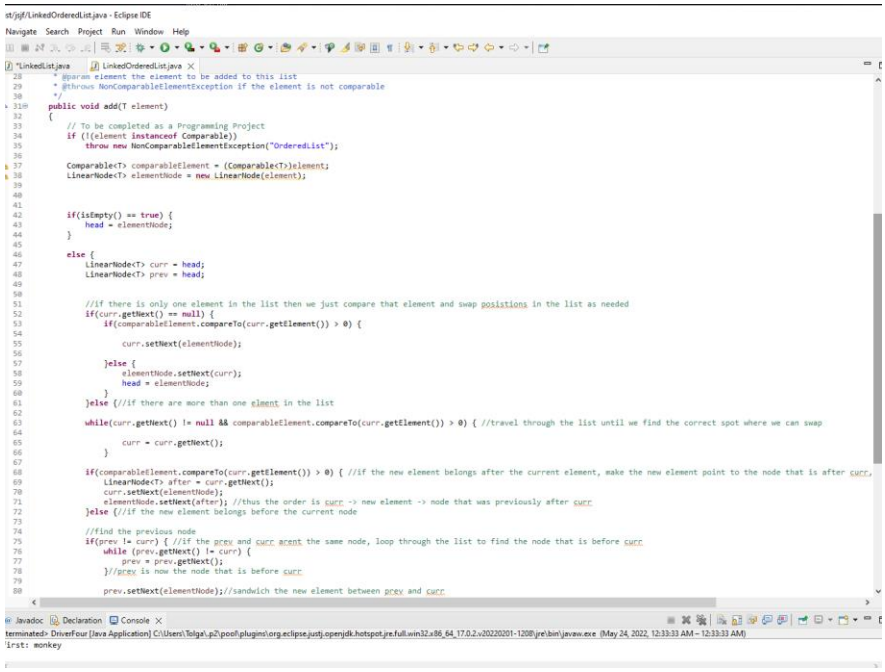
```
    /**
     * Adds the specified element to this list after the given target.
     *
     * @param  element the element to be added to this list
     * @param  target the target element to be added after
     * @throws ElementNotFoundException if the target is not found
     */
    public void addAfter(T element, T target)
    {


        if(isEmpty())
            throw new EmptyCollectionException("LinkedList");
        // To be completed as a Programming Project
        int size = size();
        LinearNode<T> elementNode = new LinearNode(element);
        LinearNode<T> curr = head;
        int countNode = 1;
        String name = target.toString();//name used for element not found exception

        while(curr != null && curr.getElement() != target) { //loop through the list to find the target
            curr = curr.getNext();
            countNode++;
        }

        if(countNode == 1) { //if the target is the head then add the element after the head and make the element point to the node that is after the head
            LinearNode<T> after = curr.getNext();
            elementNode.setNext(after);
            head.setNext(elementNode);
            count++;
            modCount++;

        }else if(countNode == size) { //if target is the tail, set the tail next to point to the element
            tail.setNext(elementNode);
            tail = elementNode;
            count++;
            modCount++;
        }else if(curr != null) {//if the target is not the head or not the tail, yet still exisits within the list
            LinearNode<T> after = curr.getNext();
            elementNode.setNext(after);
            curr.setNext(elementNode);
            count++;
            modCount++;
        }else {
            throw new ElementNotFoundException(name);//target is not in the list
        }

    }
}
```

```
adoc  Declaration  Console
ated> DriverFour [Java Application] C:\Users\Tolga\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.2.v20220201-1208\jre\bin\javaw.exe  (May 24, 2022, 12:33:33 AM – 12:33:33 AM)
: monkey
```

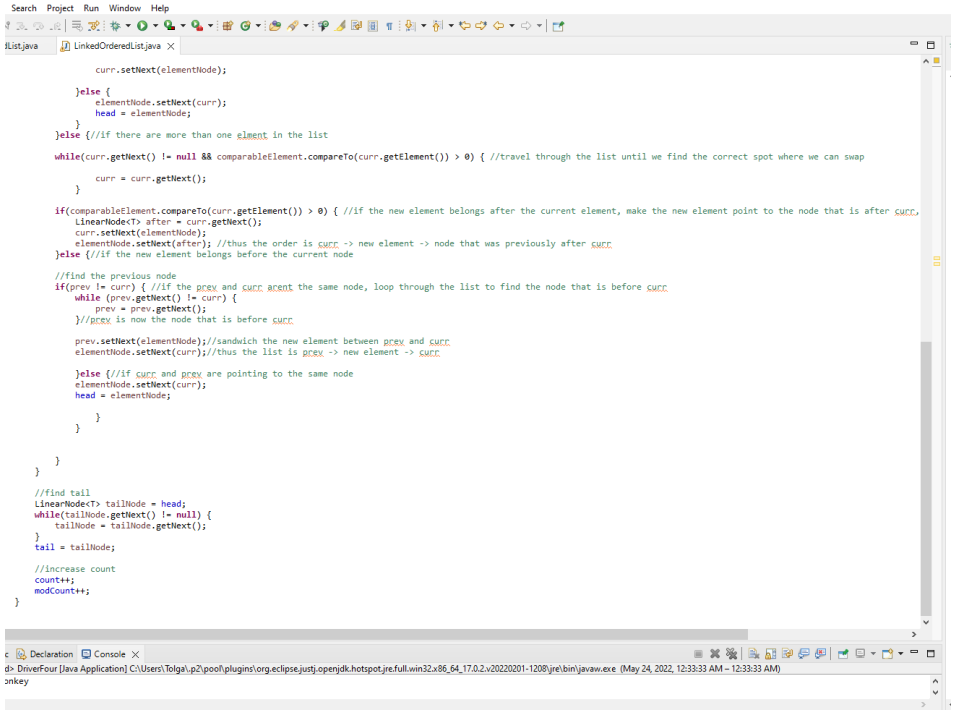
The AddAfter() function was also required to be completed as part of the programming project, however it was not used for the LinkStack.

Now let me show you the HARDEST part of the assignment WHICH was not even used for the LinkStack!:

THE LinkOrderedListClass!

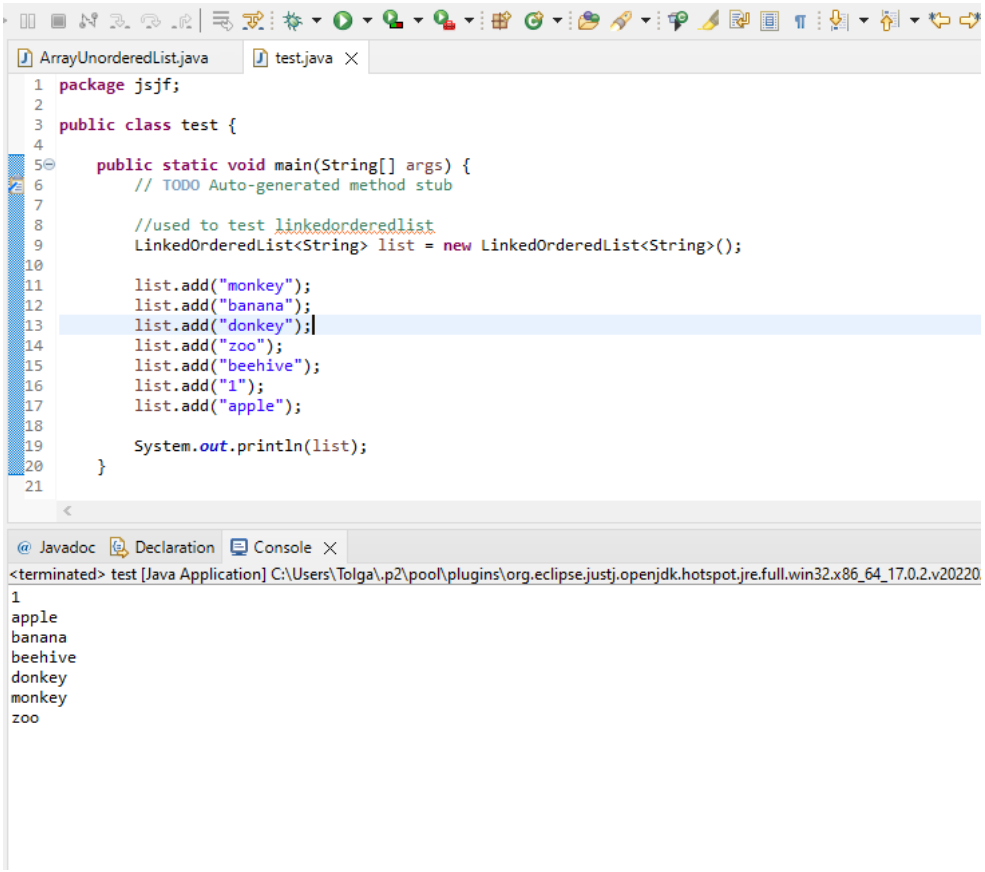


```
    * @param element the element to be added to this list
    * @throws NonComparableElementException if the element is not comparable
    */
   public void add(T element)
   {
       // To be completed as a Programming Project
       if (!(element instanceof Comparable))
           throw new NonComparableElementException("OrderedList");

       Comparable<T> comparableElement = (Comparable<T>)element;
       LinearNode<T> elementNode = new LinearNode(element);


       if(isEmpty() == true) {
           head = elementNode;
       }

       else {
           LinearNode<T> curr = head;
           LinearNode<T> prev = head;

           //if there is only one element in the list then we just compare that element and swap positions in the list as needed
           if(curr.getNext() == null) {
               if(comparableElement.compareTo(curr.getElement()) > 0) {

                   curr.setNext(elementNode);

               }else {
                   elementNode.setNext(curr);
                   head = elementNode;
               }
           }else {//if there are more than one element in the list

           while(curr.getNext() != null && comparableElement.compareTo(curr.getElement()) > 0) { //travel through the list until we find the correct spot where we can swap

                   curr = curr.getNext();
               }

           if(comparableElement.compareTo(curr.getElement()) > 0) { //if the new element belongs after the current element, make the new element point to the node that is after curr,
               LinearNode<T> after = curr.getNext();
               curr.setNext(elementNode);
               elementNode.setNext(after); //thus the order is curr -> new element -> node that was previously after curr
           }else {//if the new element belongs before the current node

           //find the previous node
           if(prev != curr) { //if the prev and curr arent the same node, loop through the list to find the node that is before curr
               while (prev.getNext() != curr) {
                   prev = prev.getNext();
               }//prev is now the node that is before curr

               prev.setNext(elementNode);//sandwich the new element between prev and curr
```



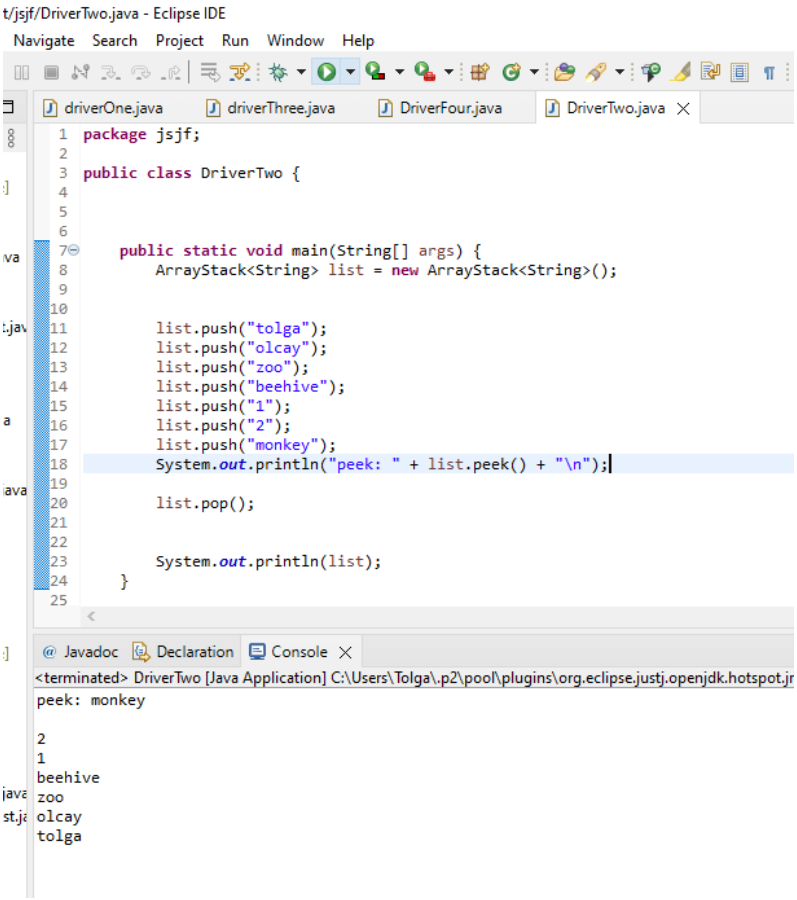
```
                   curr.setNext(elementNode);

               }else {
                   elementNode.setNext(curr);
                   head = elementNode;
               }
           }else {//if there are more than one element in the list

           while(curr.getNext() != null && comparableElement.compareTo(curr.getElement()) > 0) { //travel through the list until we find the correct spot where we can swap

                   curr = curr.getNext();
               }

           if(comparableElement.compareTo(curr.getElement()) > 0) { //if the new element belongs after the current element, make the new element point to the node that is after curr,
               LinearNode<T> after = curr.getNext();
               curr.setNext(elementNode);
               elementNode.setNext(after); //thus the order is curr -> new element -> node that was previously after curr
           }else {//if the new element belongs before the current node

           //find the previous node
           if(prev != curr) { //if the prev and curr arent the same node, loop through the list to find the node that is before curr
               while (prev.getNext() != curr) {
                   prev = prev.getNext();
               }//prev is now the node that is before curr

               prev.setNext(elementNode);//sandwich the new element between prev and curr
               elementNode.setNext(curr);//thus the list is prev -> new element -> curr

               }else {//if curr and prev are pointing to the same node
               elementNode.setNext(curr);
               head = elementNode;

                   }
               }


           }
       }

       //find tail
       LinearNode<T> tailNode = head;
       while(tailNode.getNext() != null) {
           tailNode = tailNode.getNext();
       }
       tail = tailNode;

       //increase count
       count++;
       modCount++;
   }
```

The add() function for this class was by far the hardest part. Below is a test driver used to test out the ListOrderedList because I want to show you the extent of how it works:




As you can see it places the values in the List where they belong.

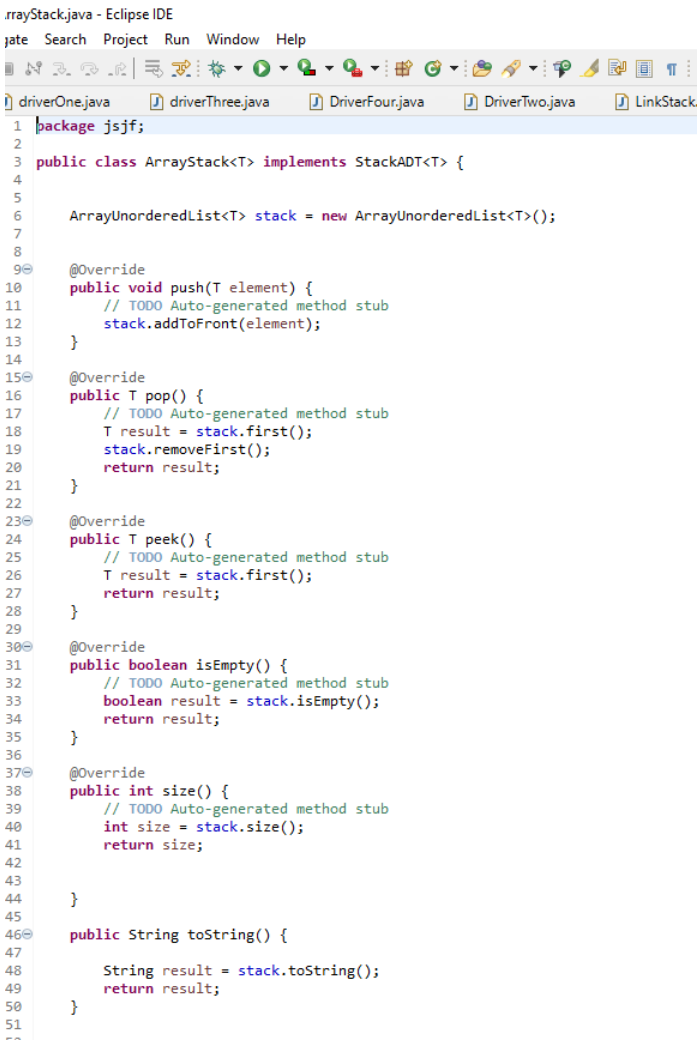
2. Stack using ArrayList




As you can see, this driver is very similar to driver one. The main difference is that instead of instead of creating a LinkStack, we created an ArrayStack! As you can see, monkey is not there when we print the stack because, it was popped.

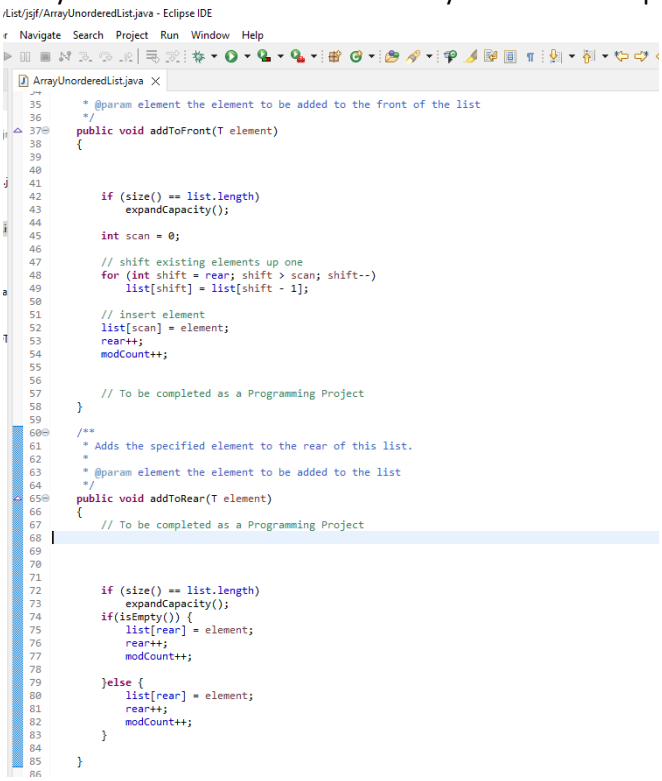
Now let me demonstrate how this works:

Lets look at the ArrayStack:

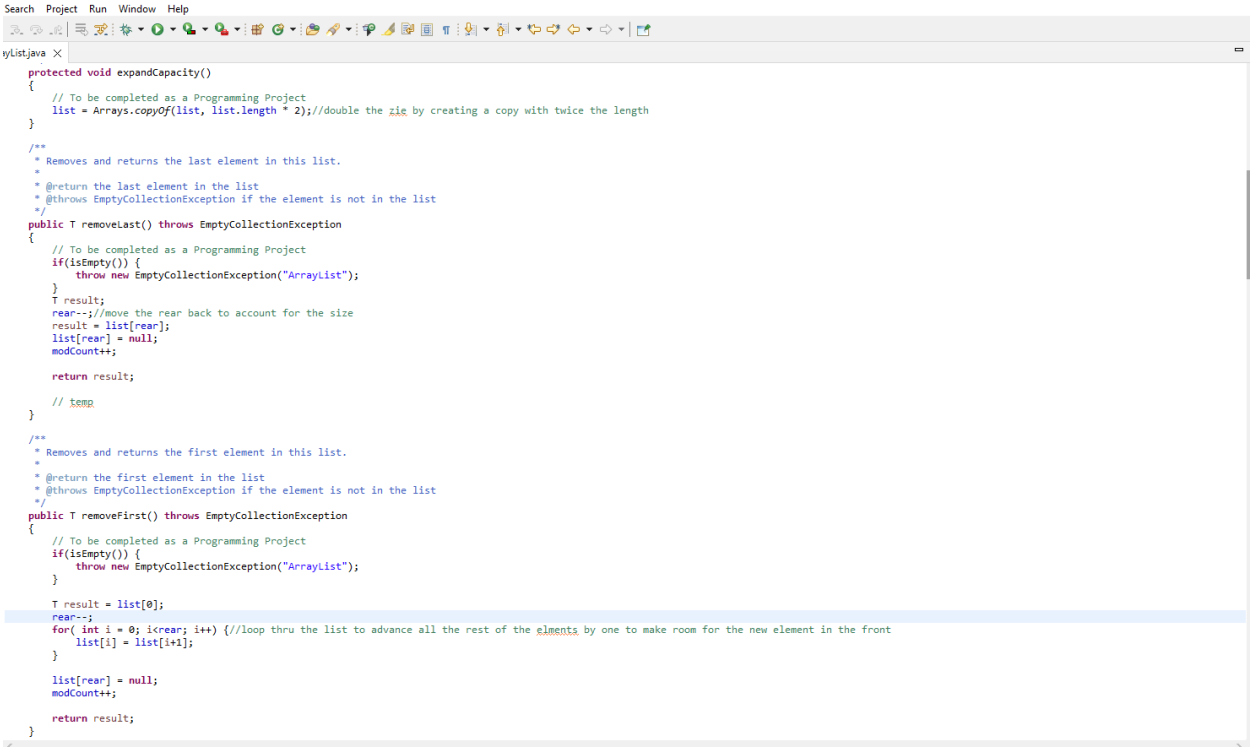



We use functions from the ArrayList and ArrayUnoredredList classes in our methods. As you can see, these methods don't really show much how these work because it is encapsulated by the ArrayUnorderedList class which extends the ArrayList Class. I will be showing you how both of those  classes work.

ArrayUnorderedList extends the ArrayList class and is part of the programming Project



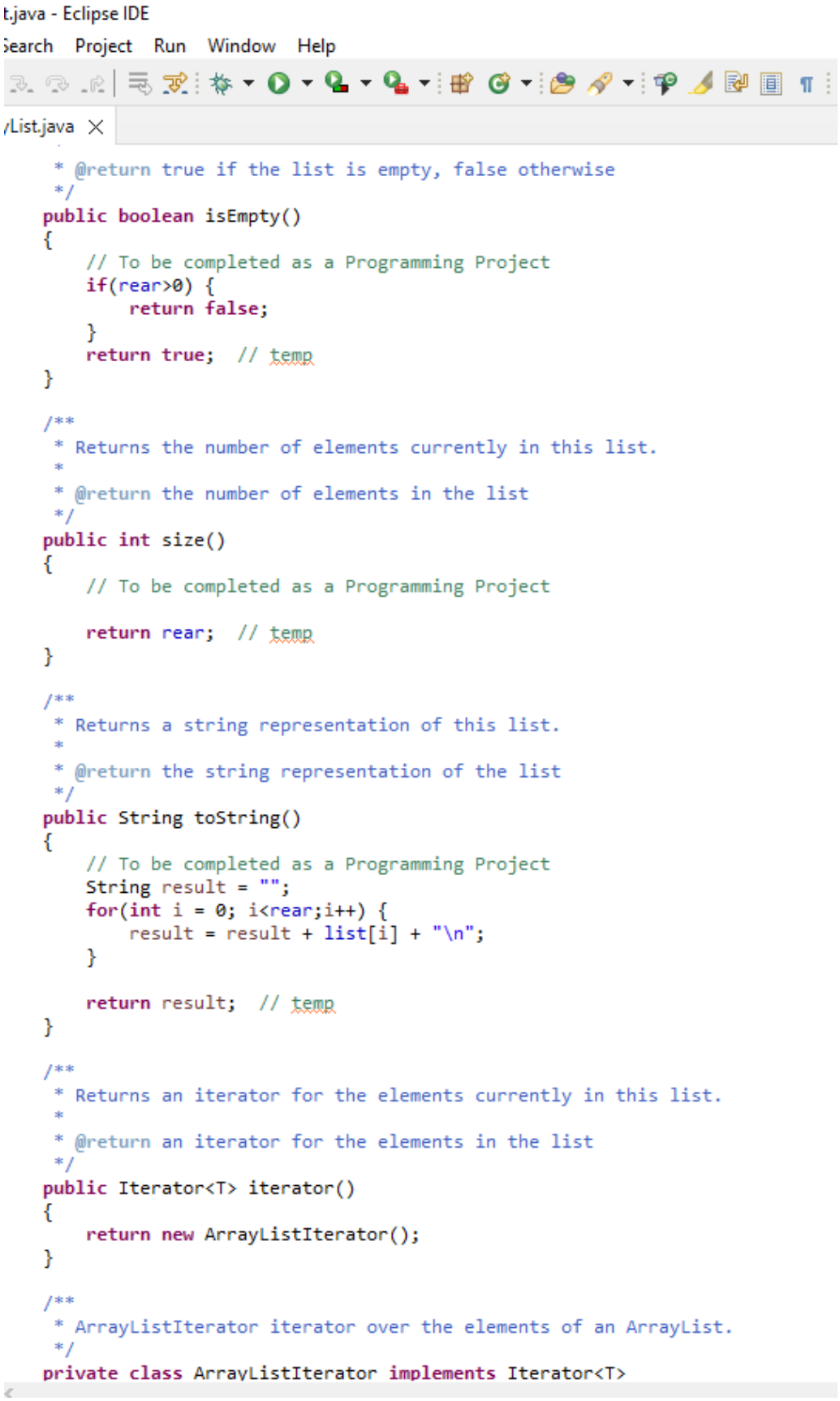

Our ArrayStack uses the addToFront method to add our objects to the front of the stack. This other functions from the ArrayUnorderedList and ArrayStack use methods from the ArrayList class:



For example, the expandcapacity function is used by functions which add items to the list, so that the list does not run out of space. If you noticed, removeFirst() is used in the pop() function in ArrayStack.

```
/**
 * Returns a reference to the element at the front of this list.
 * The element is not removed from the list.  Throws an
 * EmptyCollectionException if the list is empty.
 *
 * @return a reference to the first element in the list
 * @throws EmptyCollectionException if the list is empty
 */
public T first() throws EmptyCollectionException
{
    // To be completed as a Programming Project
    if(isEmpty()) {
        throw new EmptyCollectionException("ArrayList");
    }


    return list[0];  // temp
}

/**
 * Returns a reference to the element at the rear of this list.
 * The element is not removed from the list. Throws an
 * EmptyCollectionException if the list is empty.
 *
 * @return a reference to the last element of this list
 * @throws EmptyCollectionException if the list is empty
 */
public T last() throws EmptyCollectionException
{
    // To be completed as a Programming Project
    if(isEmpty()) {
        throw new EmptyCollectionException("ArrayList");
    }

    return list[rear-1];  // temp
}

/**
 * Returns true if this list contains the specified element.
 *
 * @param target the target element
 * @return true if the target is in the list, false otherwise
 */
public boolean contains(T target)
{
    return (find(target) != NOT_FOUND);
}
```

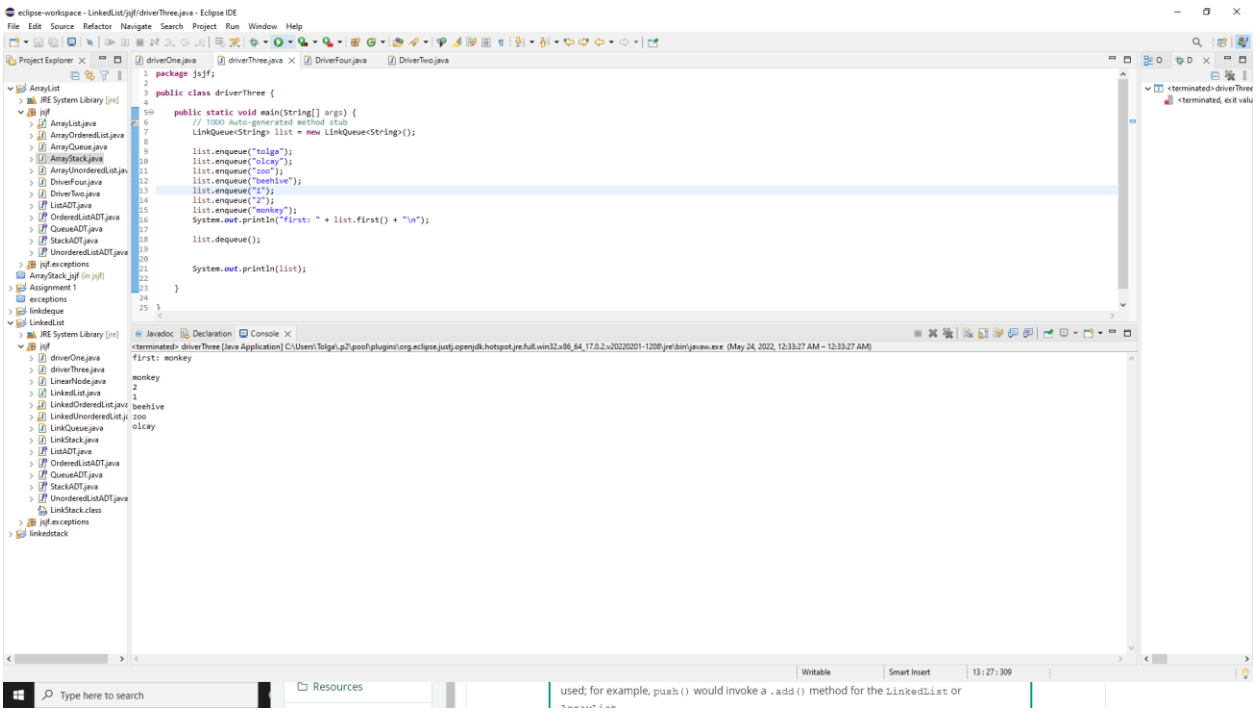
The first() function is used in the peek(), and also the pop() method in the  ArrayStack, allowing us to save the value which is in the top of the stack and return.


t.java - Eclipse IDE
Search Project Run Window Help
/List.java
* @return true if the list is empty, false otherwise
*/
public boolean isEmpty()
{
// To be completed as a Programming Project
if(rear>0) {
return false;
}
return true; // temp
}
/**
* Returns the number of elements currently in this list.
*
* @return the number of elements in the list
*/
public int size()
{
// To be completed as a Programming Project
return rear; // temp
}
/**
* Returns a string representation of this list.
*
* @return the string representation of the list
*/
public String toString()
{
// To be completed as a Programming Project
String result = "";
for(int i = 0; i<rear;i++) {
result = result + list[i] + "\n";
}
return result; // temp
}
/**
* Returns an iterator for the elements currently in this list.
*
* @return an iterator for the elements in the list
*/
public Iterator<T> iterator()
{
return new ArrayListIterator();
}
/**
* ArrayListIterator iterator over the elements of an ArrayList.
*/
private class ArrayListIterator implements Iterator<T>

The isEmpty and the toString functions from ArrayList are used in the ArrayStack to check the size of the stack and to print the stack.

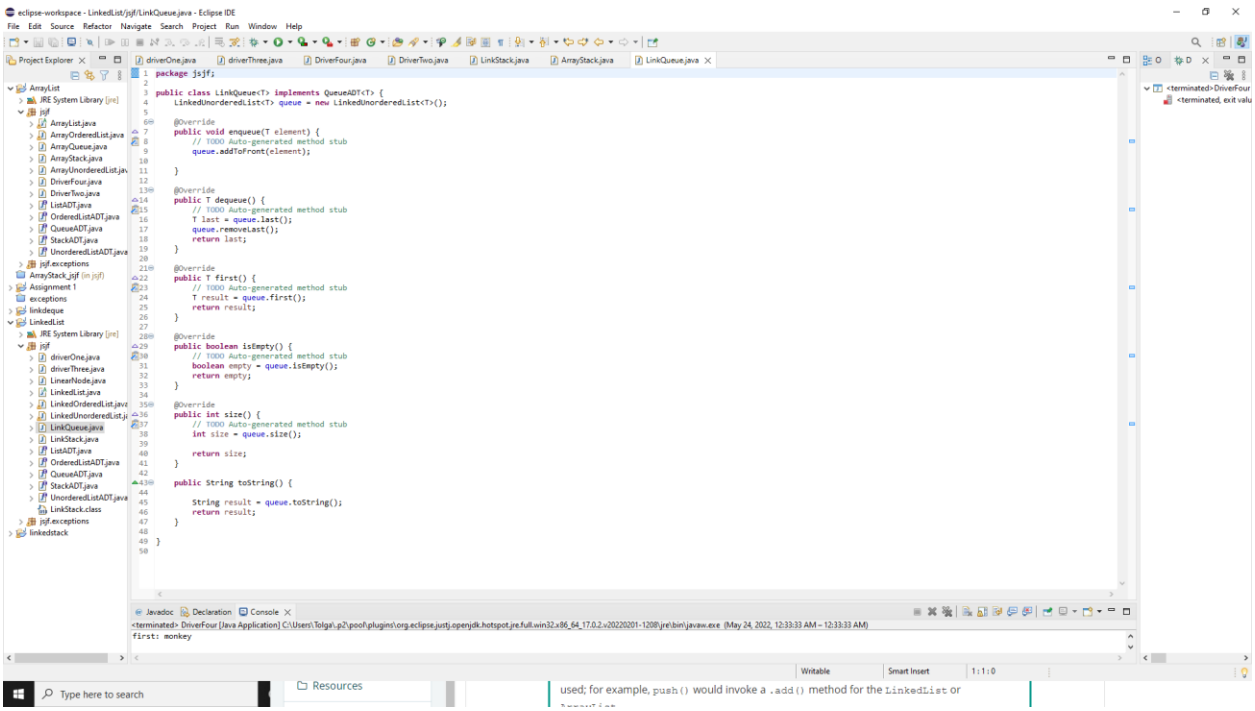
Now you understand how we can use the ArrayList and ArrayUnorderedList to create an ArrayStack.

3. Linked Queue




As you can see with our linkQueue driver, the string “tolga” is no longer in the queue when we print the queue. That’s because “tolga” was the first string that was enqueued, so when we dequeue(), tolga is removed from the queue.

Now, Lets view the LinkQueue class.

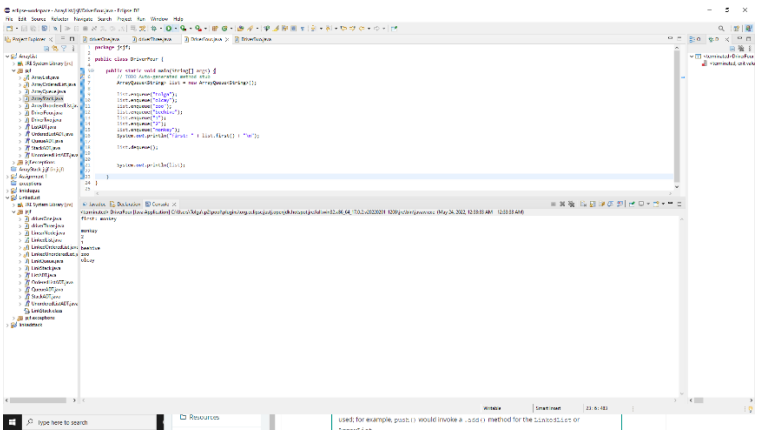



As you can see, just like LinkStack, LinkQueue uses functions from the LinkList and the LinkUnorderedList classes. I am able to re-use theses classes for my LinkStack and my LinkQueue via encapsulation.

As you can see when we dequeue, instead of using the removeFirst() function like we did for our stack, we instead do removeLast(); since by doing so , we are removing the first value added to the list.

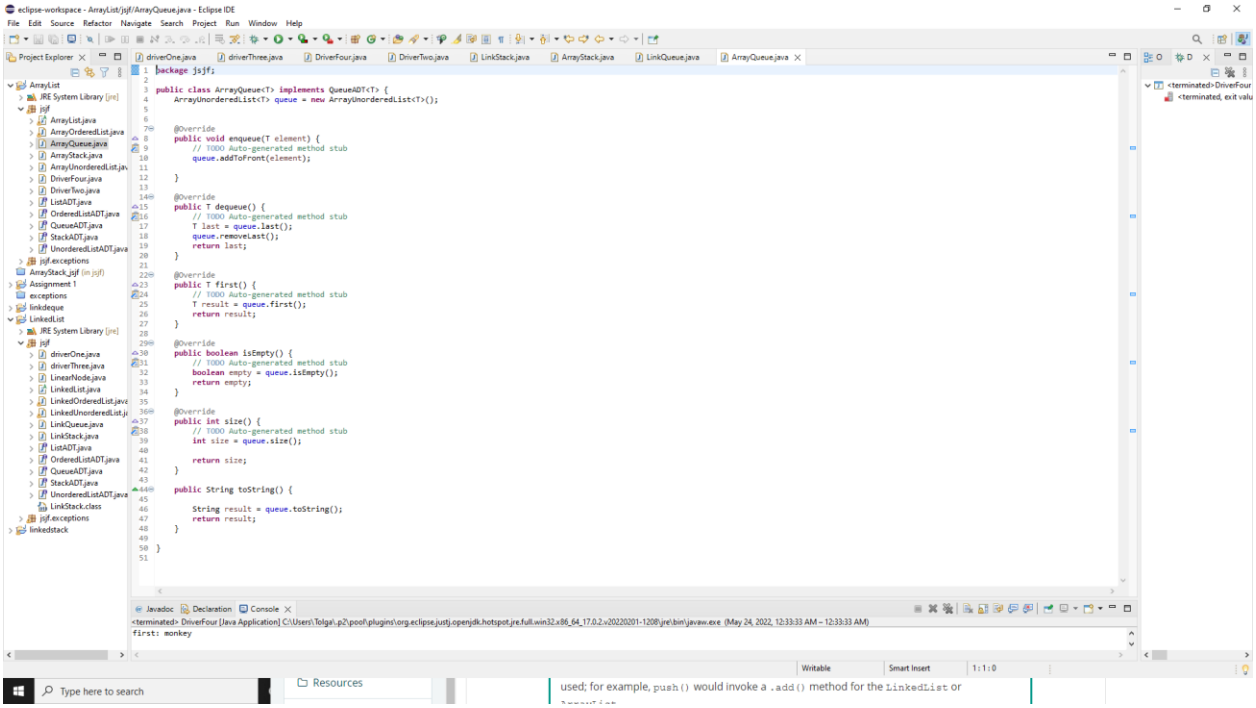
Since I already showed you how the LinkedList and the LinkedUnorderedList classes work, I will not be pasting the screenshots again. (just scroll up)

4. ArrayQueue



As you can see, driver Four creates a Arrayqueue and pushes strings onto the queue. Just like driver three, we can notice that the string “tolga” is not present when we print the queue. That’s because it was the first value to be enqueued, thus making it the first value to be dequeued.

Now lets look at the ArrayQueue class




As you can see, this is very similar to the LinkedQueue class, however the difference is that it creates an ArrayList and uses functions from the arrayList and ArrayUnorderedList to create the ArrayStack.

Since we already went into detail on how ArrayList and ArrayUnorederedList classes work, you can scroll up and compare the methods that are used for the queue.

You now have an idea of how ArrayQueue, LinkQueue, ArrayStack, and LinkStack work by implanting LinkedLists, Linkunorderedlists, and ArrayLists and ArrayUnorderedLists!