Matt Matuk

CSIT – 211

12 March 2015

Lab 5

**Programming Project PP 15.14**

-----**SPEC**-----

Create an implementation of a doubly linked DoubleOrderedList class. You will need to create a DoubleNode class, a DoubleList class, and a DoubleIterator class.

**SCRUM**

* Class DoubleOrderList extends DoubleList
  + Variables
    - Private Int count
    - Private DoubleNode<T> head
    - Priavte DoubleNode<T> tail
    - Private final static int NOT\_FOUND = -1
    - Private int modCount
  + Method
    - DoubleOrderedList()
    - Add(T element)
    - Private int find(T target)
    - Bool contains(T target)
* Class DoubleOrderListTester
  + Method
    - Main()
* Class DoubleNode<T>
  + Variables
    - Private DoubleNode<T> next
    - Private DoubleNode<T> previous
    - Private T element
  + Methods
    - DoubleNode()
    - DoubleNode(T elem)
    - setNext(DoubleNode<T> node)
    - setPrevious(DoubleNode<T> node)
    - DoubleNode<T> getNext()
    - DoubleNode<T> getPrevious()
    - T getElement()
    - setElement(T elem)
* Class DoubleList
  + Methods
    - T removeFirst()
    - T removeLast()
    - T remove(T element)
    - T first()
    - T last()
    - Bool isEmpty()
    - Int size()
    - Iterator iterator()
    - String toString()
  + Class Private DoubleIterator
    - Variable
      * Private int iteratorModCount
      * Private DoubleNode<T> current
    - Methods
      * DoubleIterator()
      * Bool hasNext()
      * T next()
      * Remove()

**Class: DoubleOrderListTester**

1. This class will test the implementation of the double linked ordered list class
2. This class will create a couple of items for the list and then make some changes on those items to show that the implementation of the double linked ordered list works

**Import**

**Process**

* Methods
  + Main()
    - Print some intro statements about the program
    - Create a new DoubleLinkedList object and add some items to the list
    - Make some changes to the items in the list
    - Print some ending statements about the program

**Class: DoubleOrderList extends DoubleList**

1. This class extends the DoubleList class and adds the add method to the class.
2. The add method will add items to the list in the proper order
3. All methods and variables are generic
4. All methods are public unless otherwise stated

**Import**

**Process**

* Method
  + DoubleOrderedList()
    - Count = 0
    - Head = tail = null
  + Add(T element)
    - If the element is not comparable
      * Throw NonComparableElementExcepetion(“OrderedList”)
    - Create a new DoubleNode newNode and set the element to the passed element
    - Create a new Comparable object comparableElement
    - Create a new DoubleNode currentNode = head
    - Int scan = 0
    - Find the insertion point by looping while scan is less than count and while the comparableElment when comparaed to the current nodes element is greater than 0
      * currentNode = currentNode’s next Node
    - Create a new DoubleNode previous set equal to currentNode getPrevious()
    - newNode setPrevious() to previous
    - newNode setNext() to currentNode
    - previous setNext() to newNode()
    - currentNode setPrevious() to newNode()
    - increase mod count

**Class: DoubleNode<T>**

1. This class is a double node that will contains a reference to the next and previous nodes.
2. This class will also contain a element
3. All methods and variables are generic

**Import**

**Process**

* Variables
  + Protected DoubleNode<T> next
  + protected DoubleNode<T> previous
  + protected T element
* Methods
  + DoubleNode()
    - Element = null
    - Next = previous = null
  + DoubleNode(T elem)
    - Element = elem
    - Next = previous = null
  + setNext(DoubleNode<T> node)
    - next = node
  + setPrevious(DoubleNode<T> node)
    - previous = node
  + DoubleNode<T> getNext()
    - return next
  + DoubleNode<T> getPrevious()
    - Return previous
  + T getElement()
    - Return element
  + setElement(T elem)
    - element = elem

**Class: DoubleList<T>**

1. This class will contain all the methods needed to create a double linked list
2. The class will be the parent class for DoubleOrderList
3. All methods are public and void unless otherwise stated
4. All methods and variables are generic unless otherwise stated

**Import**

**Process**

* Variables
  + Private Int count
  + Private DoubleNode<T> head
  + Priavte DoubleNode<T> tail
  + Private int modCount
* Methods
  + T removeFirst()
    - Need to return the head element
    - Set head equal to head.getNext()
  + T removeLast()
    - Need to return the tail element
    - Set the tail equal to tail.getPrevious()
  + T remove(T element)
    - Need to return the element removed
    - Use the find() method to check to see if the element is in the list
    - Throw exception if not found
    - If the element is found
      * Rest the nodes before and after to point to the new nodes
      * Decrease count by 1
      * Increase modCount by 1
  + T first()
    - Return head
  + T last()
    - Return tail
  + Bool isEmpty()
    - Return true if count = 0
  + Int size()
    - Return count
  + Iterator iterator()
  + Private DoubleNode<T> find(T target)
    - Need int scan = 0 and DoubleNode<T> result = null
    - If the list is not empty
      * Loop while result is equal to null and scan is less than the size of the list
        + If the target equals the element

Result = element

* + - * + Else

Scan increase by one

* + - Return result
  + Bool contains(T target)
    - Return the result of (find(target) != null)
  + String toString()
* Class Private DoubleIterator
  + Variable
    - Private int iteratorModCount
    - Private DoubleNode<T> current
  + Methods
    - DoubleIterator()
      * Current = head
      * iteratorModCount = modCount
    - Bool hasNext()
      * If the mod counts are different
        + Throw exception
      * Return (current != null)
    - T next()
      * If hasNext is false
        + Throw excepetion
      * T result = current’s element
      * Set current equal to current’s next node
      * Return result
    - Remove()
      * Not supported

**CODE**

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// DoubleOrderListTester.java Matt Matuk

// CSIT 211 Project 15.14 pg 578

// 1. This class will test the implementation of the double linked

// ordered list class

// 2. This class will create a couple of items for the list and then

// make some changes on those items to show that the implementation of

// the double linked ordered list works

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**public** **class** DoubleOrderListTester

{

**public** **static** **void** main(String[] args) **throws** Exception

{

DoubleOrderList<String> test = **new** DoubleOrderList<String>();

System.***out***.println("Welcome to my application.\n"

+ "I will be showing you a implemented Double Linked Ordered"

+ "List.");

System.***out***.println();

test.add("Zoo");

test.add("Abc");

test.add("Ball");

test.add("#123");

test.add("I want a cookie.");

test.add("Something else.");

test.add("I do not know");

test.add("Testing");

System.***out***.println("\*\*Before Changes\*\*\n"

+ "---------------------------------\n"

+ "Here is the contents of the list:\n"

+ test

+ "\n---------------------------------");

test.remove("I want a cookie.");

test.removeFirst();

System.***out***.println();

System.***out***.println("\*\*Removed \"I want a cookie.\" and removed first item.\*\*\n"

+ "---------------------------------\n"

+ "Here is the contents of the list:\n"

+ test

+ "\n---------------------------------");

test.add("Something else");

test.add("This is an element in the list");

System.***out***.println();

System.***out***.println("\*\*Added two new Strings\*\*\n"

+ "---------------------------------\n"

+ "Here is the contents of the list:\n"

+ test

+ "\n---------------------------------");

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// DoubleOrderList.java Matt Matuk

// CSIT 211 Project 15.14 pg 578

// 1. This class extends the DoubleList class and adds the add

// method to the class.

// 2. The add method will add items to the list in the proper order

// 3. All methods and variables are generic

// 4. All methods are public unless otherwise stated

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**public** **class** DoubleOrderList<T> **extends** DoubleList<T>

{

**public** DoubleOrderList()

{

count = 0;

head = tail = **null**;

}

/\*\*

\* Adds a element to the list and places the element in the proper

\* order. Keeps the list ordered.

\*

\* **@param** element The element to be added

\* **@throws** Exception Thrown of the element is not comparable

\*/

**public** **void** add(T element) **throws** Exception

{

**if** (!(element **instanceof** Comparable))

{

**throw** **new** Exception("Element is not comparable");

}

// this is for if the list is empty

**if** (isEmpty())

{

head = **new** DoubleNode<>(element);

tail = head;

}

**else**

{

Comparable<T> comparableElement = (Comparable<T>)element;

DoubleNode<T> previous = **null**;

DoubleNode<T> current = head;

DoubleNode<T> newNode = **new** DoubleNode<>(element);

**int** scan = 0;

// loops until the element goes before the current element

// being checked

**while** ((scan < count && current != **null**) && comparableElement.compareTo(current.getElement()) > 0)

{

previous = current;

current = current.getNext();

}

**if** (current == head) // if the element goes at the front

{

head.setPrevious(newNode);

newNode.setNext(head);

head = newNode;

}

**else** **if** (current == **null**) // if the element goes at the end

{

tail.setNext(newNode);

newNode.setPrevious(tail);

tail =newNode;

}

**else**

{

previous.setNext(newNode);

newNode.setPrevious(previous);

newNode.setNext(current);

current.setPrevious(newNode);

}

}

count++;

modCount++;

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// DoubleList.java Matt Matuk

// CSIT 211 Project 15.14 pg 578

// 1. This class will contain all the methods needed to create a

// double linked list

// 2. The class will be the parent class for DoubleOrderList

// 3. All methods are public and void unless otherwise stated

// 4. All methods and variables are generic unless otherwise stated

// 5. This class will also contain a private class for the Iterator

// 6. This class will be adstract

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**import** java.util.ConcurrentModificationException;

**import** java.util.Iterator;

**import** java.util.NoSuchElementException;

**public** **abstract** **class** DoubleList<T> **implements** Iterable<T>

{

**protected** **int** count;

**protected** DoubleNode<T> head, tail;

**protected** **int** modCount;

/\*\*

\* Removes the first element

\*

\* **@return** The element that will be removed

\*/

**public** T removeFirst()

{

**if** (!isEmpty())

{

T result = head.getElement();

DoubleNode<T> tmp = head.getNext();

tmp.setPrevious(**null**);

head = tmp;

count--;

modCount++;

**return** result;

}

**else**

{

**return** **null**;

}

}

/\*\*

\* Removes the last element

\*

\* **@return** The element removed

\*/

**public** T removeLast()

{

**if** (!isEmpty())

{

T result = tail.getElement();

DoubleNode<T> tmp = tail.getPrevious();

tmp.setNext(**null**);

tail = tmp;

count--;

modCount++;

**return** result;

}

**else**

{

**return** **null**;

}

}

/\*\*

\* Removes a certain element

\*

\* **@exception** Throws exception if the element was not found

\* **@param** element The element to be removed

\* **@return** The element that was removed

\*/

**public** T remove(T element)

{

DoubleNode<T> tmp = find(element);

**if** (tmp == **null**)

{

**throw** **new** NoSuchElementException("DoubleList");

}

**else**

{

DoubleNode<T> previous, next;

**if** (tmp == head) // if element = the first element

{

next = head.getNext();

next.setPrevious(**null**);

head = next;

}

**else** **if** (tmp == tail) // if element = the last element

{

previous = tail.getPrevious();

previous.setNext(**null**);

tail = previous;

}

**else**

{

previous = tmp.getPrevious();

next = tmp.getNext();

previous.setNext(next);

next.setPrevious(previous);

}

count--;

modCount++;

**return** tmp.getElement();

}

}

/\*\*

\*

\* **@return** The first element

\*/

**public** T first()

{

**return** head.getElement();

}

/\*\*

\*

\* **@return** The last element

\*/

**public** T last()

{

**return** tail.getElement();

}

/\*\*

\*

\* **@return** True if empty

\*/

**public** **boolean** isEmpty()

{

**return** (count == 0);

}

/\*\*

\*

\* **@return** The size of teh collection

\*/

**public** **int** size()

{

**return** count;

}

/\*\*

\* Creates a iterator for the collection

\*

\* **@return** the iterator created

\*/

**public** Iterator<T> iterator()

{

**return** **new** DoubleIterator();

}

/\*\*

\* Find a certain element

\*

\* **@param** target Element to be found

\* **@return** The node which contains the element

\*/

**public** DoubleNode<T> find(T target)

{

**int** scan = 0;

DoubleNode<T> result = **null**;

DoubleNode<T> current = head;

**if** (!isEmpty())

{

**while** (result == **null** && current != **null**)

{

**if** (current.getElement().equals(target))

{

result = current;

}

**else**

{

current = current.getNext();

}

}

}

**return** result;

}

/\*\*

\* Checks to see if a certain element is in the collection

\*

\* **@param** target The target element

\* **@return** True if found

\*/

**public** **boolean** contains(T target)

{

**return** (find(target) != **null**);

}

/\*\*

\* Converts the last to Strings

\* **@return** The String which contains all the elements in the collection

\*/

**public** String toString()

{

String result = "";

**if** (count == 0)

{

result = "Empty";

}

**else**

{

DoubleNode<T> tmp = head.getNext();

result = (String)head.getElement();

**while** (tmp.getNext() != **null**)

{

result = result + "\n" + tmp.getElement();

tmp = tmp.getNext();

}

result = result + "\n" + tmp.getElement();

}

**return** result;

}

/\*\*

\* A private class that contains the iterator for this collection

\* **@author** Matt

\*

\*/

**private** **class** DoubleIterator **implements** Iterator<T>

{

**private** **int** iteratorModCount;

**private** DoubleNode<T> current;

/\*\*

\* Create the iterator

\*/

**public** DoubleIterator()

{

current = head;

iteratorModCount = modCount;

}

/\*\*

\* Checks to see if the iterator has another item.

\*

\* **@return** True if there is another item

\*/

**public** **boolean** hasNext() **throws** ConcurrentModificationException

{

**if** (iteratorModCount != modCount)

{

**throw** **new** ConcurrentModificationException();

}

**return** (current != **null**);

}

/\*\*

\* returns the next item in the iteration

\*/

**public** T next() **throws** NoSuchElementException

{

**if** (hasNext() == **false**)

{

**throw** **new** NoSuchElementException();

}

T result = current.getElement();

current = current.getNext();

**return** result;

}

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// DoubleNode.java Matt Matuk

// CSIT 211 Project 15.14 pg 578

// 1. This class is a double node that will contains a reference

// to the next and previous nodes.

// 2. This class will also contain a element

// 3. All methods and variables are generic

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**public** **class** DoubleNode<T>

{

**private** DoubleNode<T> next, previous;

**private** T element;

/\*\*

\* Creates a Double linked node with a element

\*/

**public** DoubleNode()

{

element = **null**;

next = previous = **null**;

}

/\*\*

\* Creates a double linked node with a passed element

\*

\* **@param** elem The element to be set for this node

\*/

**public** DoubleNode(T elem)

{

element = elem;

next = previous = **null**;

}

/\*\*

\* Sets the next node

\*

\* **@param** node Node to be set

\*/

**public** **void** setNext(DoubleNode<T> node)

{

next = node;

}

/\*\*

\* Sets the previous node

\*

\* **@param** node The node to be set

\*/

**public** **void** setPrevious(DoubleNode<T> node)

{

previous = node;

}

/\*\*

\* Gets the next node

\*

\* **@return** The next node

\*/

**public** DoubleNode<T> getNext()

{

**return** next;

}

/\*\*

\* Gets the previous node

\*

\* **@return** The previous node

\*/

**public** DoubleNode<T> getPrevious()

{

**return** previous;

}

/\*\*

\* Gets the element for this node

\*

\* **@return** The element

\*/

**public** T getElement()

{

**return** element;

}

/\*\*

\* Sets the element for this node

\*

\* **@param** elem Element to be set

\*/

**public** **void** setElement(T elem)

{

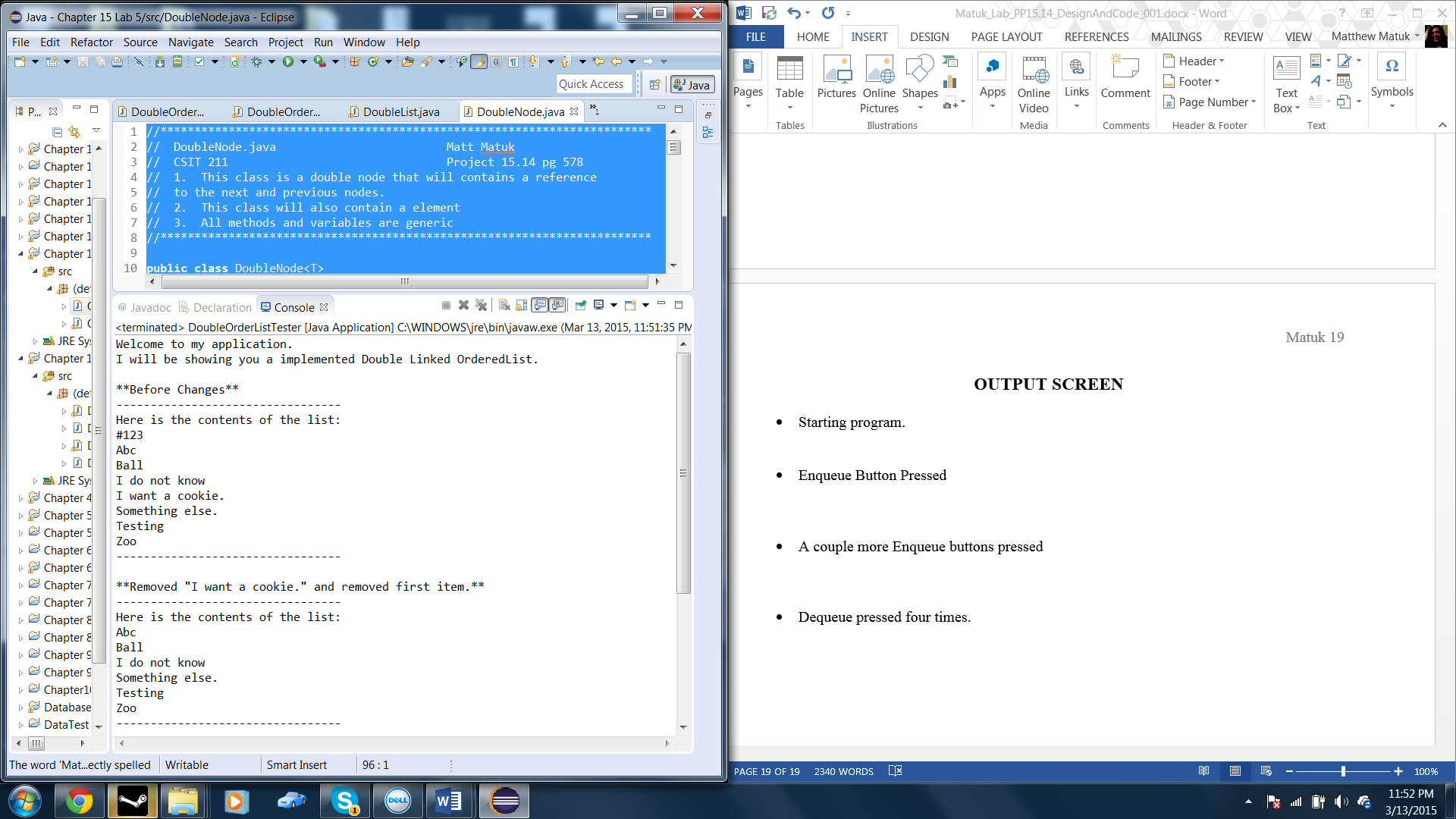
element = elem;

}

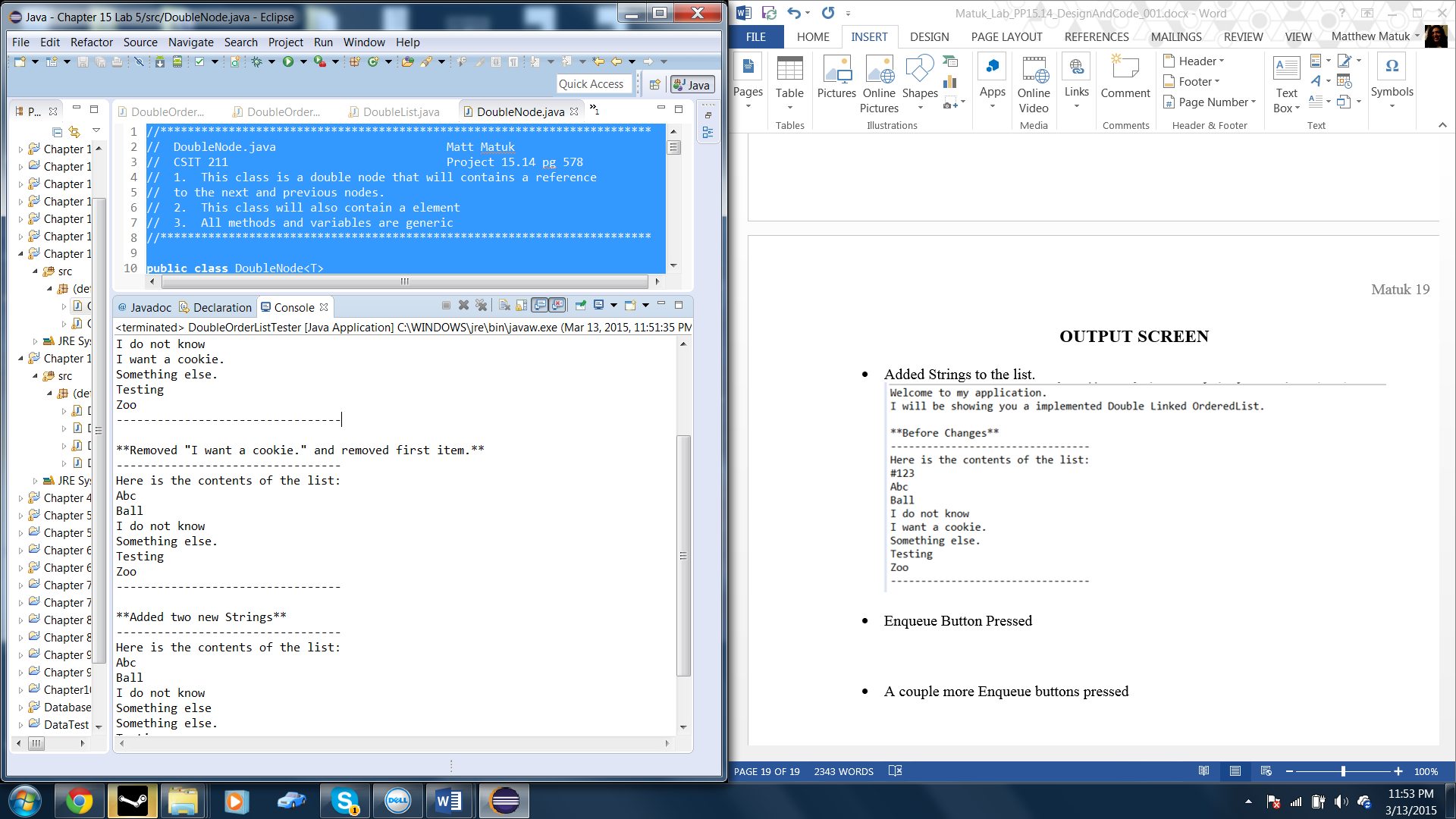
}

**OUTPUT SCREEN**

* Added Strings to the list.



* Removed two strings from the list



* Added two new strings to the list

