Algorithms, Processes and Data

LOGBOOK

WAQAS MUSHARAF (U1561634)

Contents

Tutorials One and Two

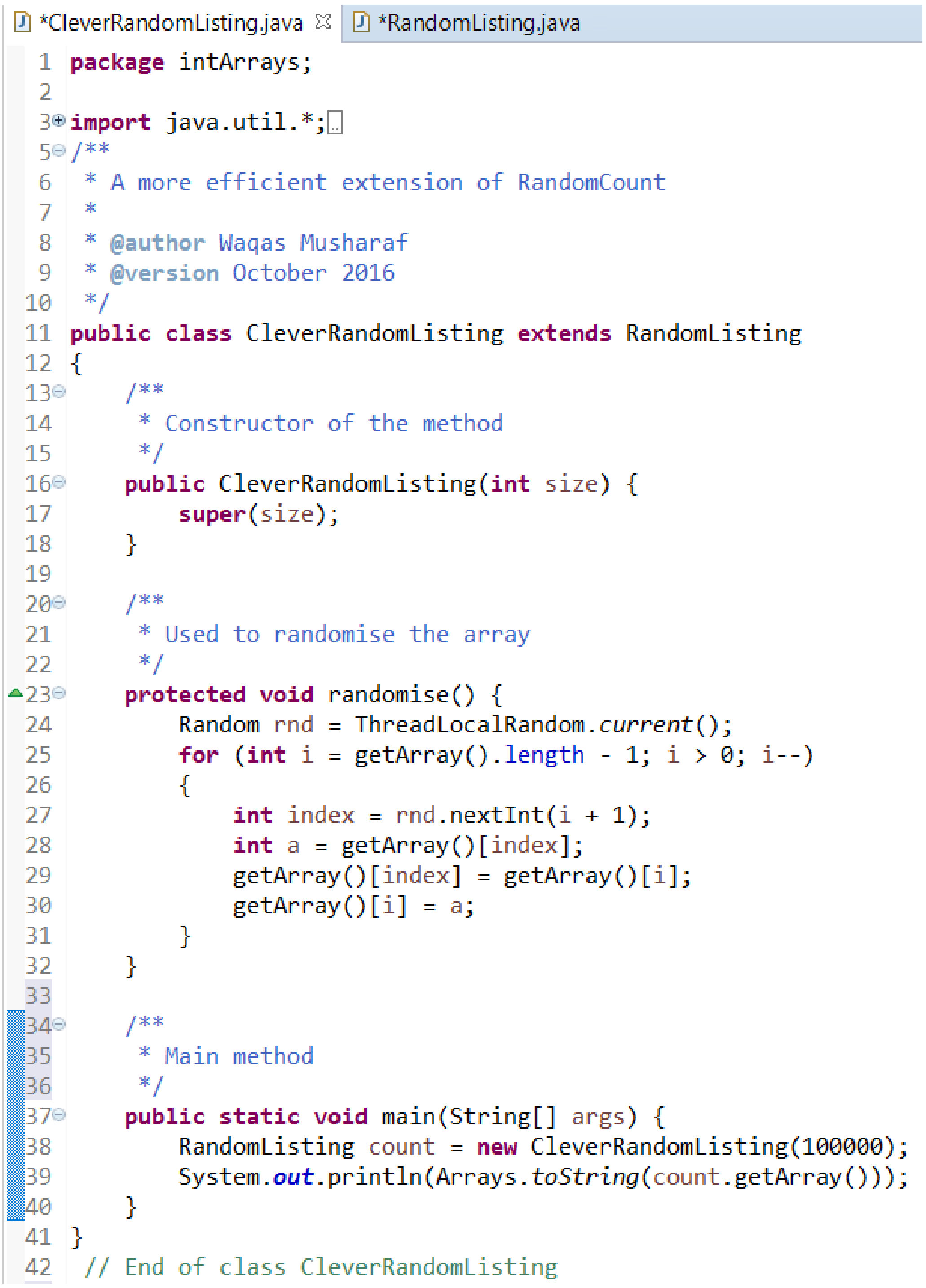
Logbook Exercises:

1) SimpleRandomListing is not an efficient implementation of the abstract RandomListing class. Design and implement a better solution. Call this class CleverRandomListing.

2) Add a timing method to the RandomListing class, and use this to compare the efficiency of the two extensions of this class.

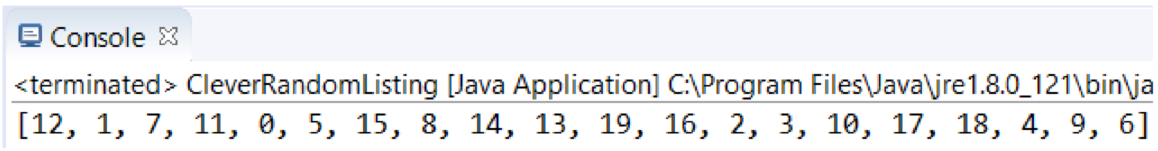
Answers:

1a) CleverRandomListing.java class:



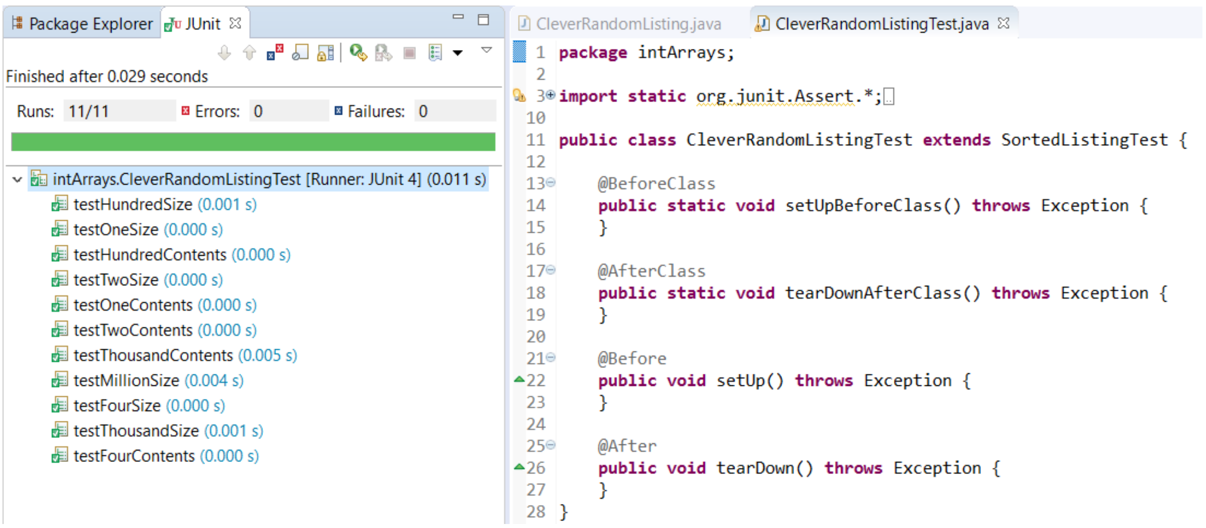
```java
package intArrays;

import java.util.*;
/**
 * A more efficient extension of RandomCount
 *
 * @author Waqas Musharaf
 * @version October 2016
 */
public class CleverRandomListing extends RandomListing
{
    /**
     * Constructor of the method
     */
    public CleverRandomListing(int size) {
        super(size);
    }

    /**
     * Used to randomise the array
     */
    protected void randomise() {
        Random rnd = ThreadLocalRandom.current();
        for (int i = getArray().length - 1; i > 0; i--)
        {
            int index = rnd.nextInt(i + 1);
            int a = getArray()[index];
            getArray()[index] = getArray()[i];
            getArray()[i] = a;
        }
    }

    /**
     * Main method
     */
    public static void main(String[] args) {
        RandomListing count = new CleverRandomListing(100000);
        System.out.println(Arrays.toString(count.getArray()));
    }
}
 // End of class CleverRandomListing
```

1b) Console output upon running CleverRandomListing.java with array size 20 (for ease of reading):



```
<terminated> CleverRandomListing [Java Application] C:\Program Files\Java\jre1.8.0_121\bin\ja
[12, 1, 7, 11, 0, 5, 15, 8, 14, 13, 19, 16, 2, 3, 10, 17, 18, 4, 9, 6]
```

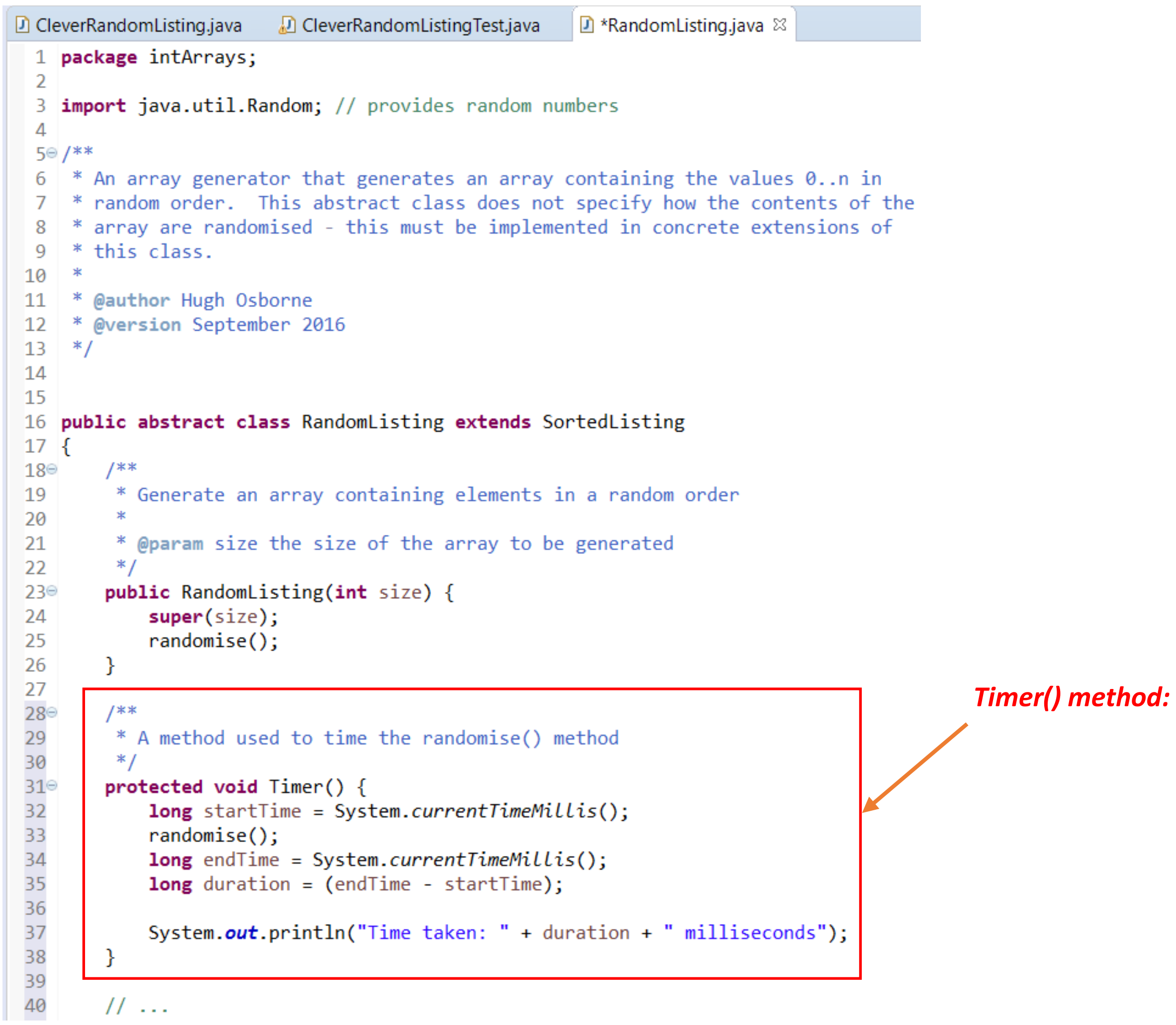
1c) JUnit tests for CleverRandomListing.java class, using CleverRandomListingTest.java:



```
Finished after 0.029 seconds
Runs: 11/11     Errors: 0     Failures: 0

intArrays.CleverRandomListingTest [Runner: JUnit 4] (0.011 s)
    testHundredSize (0.001 s)
    testOneSize (0.000 s)
    testHundredContents (0.000 s)
    testTwoSize (0.000 s)
    testOneContents (0.000 s)
    testTwoContents (0.000 s)
    testThousandContents (0.005 s)
    testMillionSize (0.004 s)
    testFourSize (0.000 s)
    testThousandSize (0.001 s)
    testFourContents (0.000 s)
```

```java
package intArrays;

import static org.junit.Assert.*;

public class CleverRandomListingTest extends SortedListingTest {

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
    }

    @AfterClass
    public static void tearDownAfterClass() throws Exception {
    }

    @Before
    public void setUp() throws Exception {
    }

    @After
    public void tearDown() throws Exception {
    }
}
```

2) The addition of the 'Timer' method in the RandomListing.java class:



```java
package intArrays;

import java.util.Random; // provides random numbers

/**
 * An array generator that generates an array containing the values 0..n in
 * random order.  This abstract class does not specify how the contents of the
 * array are randomised - this must be implemented in concrete extensions of
 * this class.
 *
 * @author Hugh Osborne
 * @version September 2016
 */


public abstract class RandomListing extends SortedListing
{
    /**
     * Generate an array containing elements in a random order
     *
     * @param size the size of the array to be generated
     */
    public RandomListing(int size) {
        super(size);
        randomise();
    }

    /**
     * A method used to time the randomise() method
     */
    protected void Timer() {
        long startTime = System.currentTimeMillis();
        randomise();
        long endTime = System.currentTimeMillis();
        long duration = (endTime - startTime);

        System.out.println("Time taken: " + duration + " milliseconds");
    }

    // ...
```

Tutorials Three and Four

Logbook Exercise:

1) Swap:

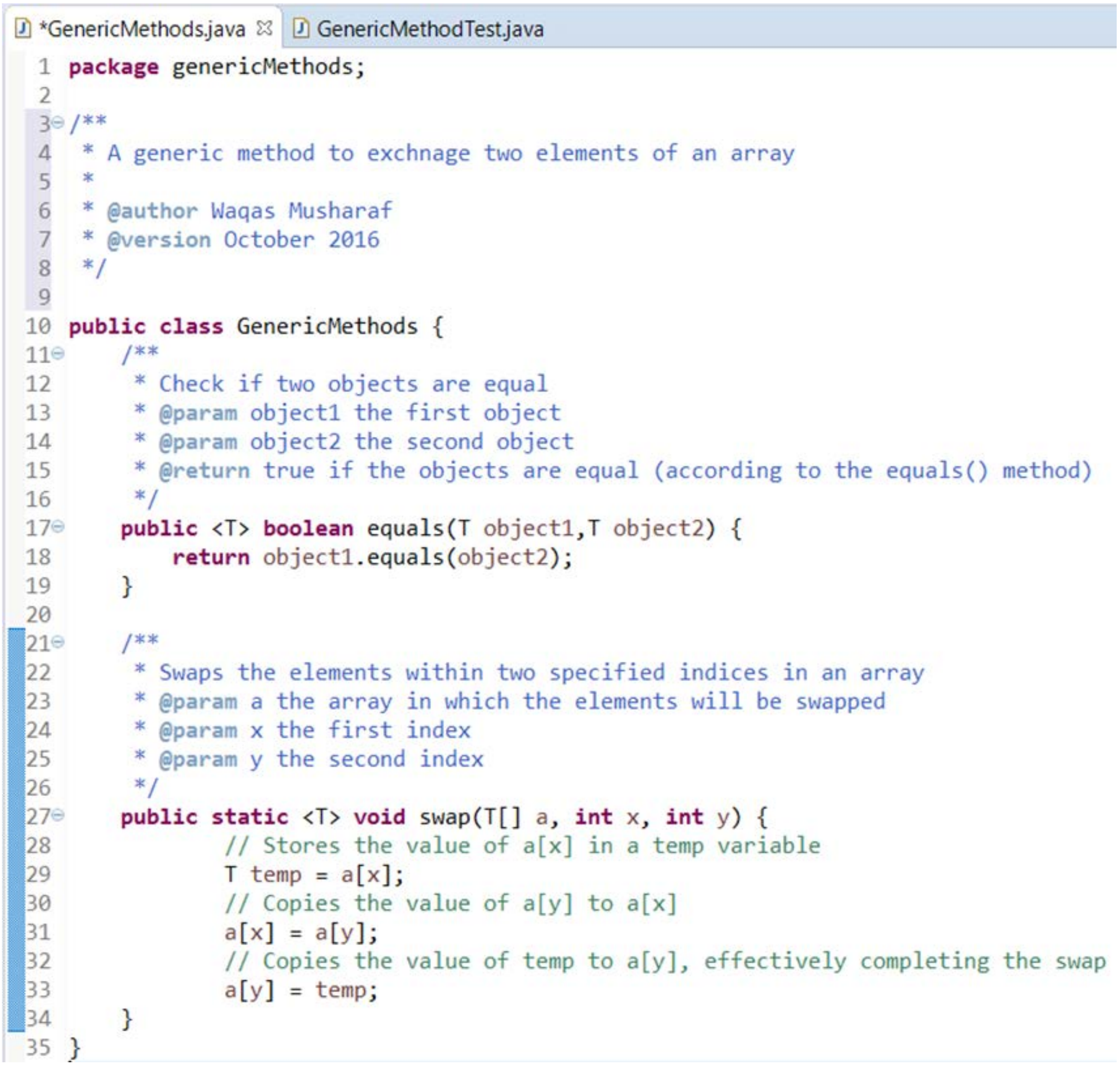
Write a generic method to exchange two elements of an array. The method should take an array, and two integer indices into the array, and swap the two entries in the array at those indices.

For example, if names is the String[] array:
{"Hugh","Andrew","Ebrahim","Diane","Paula","Simon"}

then, after a call of swap(names,1,4), the array names should contain
{"Hugh","Paula","Ebrahim","Diane","Andrew","Simon"}

Answers:

1a) GenericMethods.java class:

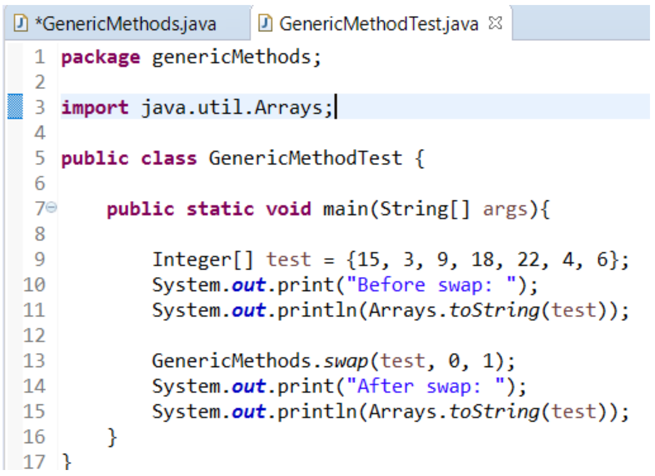


*GenericMethods.java   GenericMethodTest.java

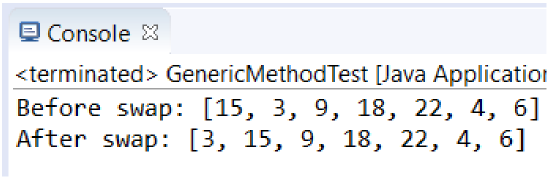
```java
package genericMethods;

/**
 * A generic method to exchnage two elements of an array
 *
 * @author Waqas Musharaf
 * @version October 2016
 */

public class GenericMethods {
    /**
     * Check if two objects are equal
     * @param object1 the first object
     * @param object2 the second object
     * @return true if the objects are equal (according to the equals() method)
     */
    public <T> boolean equals(T object1,T object2) {
        return object1.equals(object2);
    }

    /**
     * Swaps the elements within two specified indices in an array
     * @param a the array in which the elements will be swapped
     * @param x the first index
     * @param y the second index
     */
    public static <T> void swap(T[] a, int x, int y) {
            // Stores the value of a[x] in a temp variable
            T temp = a[x];
            // Copies the value of a[y] to a[x]
            a[x] = a[y];
            // Copies the value of temp to a[y], effectively completing the swap
            a[y] = temp;
    }
}
```

continued on the next page

1b) GenericMethodTest.java class and console output:



```java
package genericMethods;

import java.util.Arrays;

public class GenericMethodTest {

    public static void main(String[] args){

        Integer[] test = {15, 3, 9, 18, 22, 4, 6};
        System.out.print("Before swap: ");
        System.out.println(Arrays.toString(test));

        GenericMethods.swap(test, 0, 1);
        System.out.print("After swap: ");
        System.out.println(Arrays.toString(test));
    }
}
```



```
<terminated> GenericMethodTest [Java Applicatio
Before swap: [15, 3, 9, 18, 22, 4, 6]
After swap: [3, 15, 9, 18, 22, 4, 6]
```

Tutorial Five

Logbook Exercises:

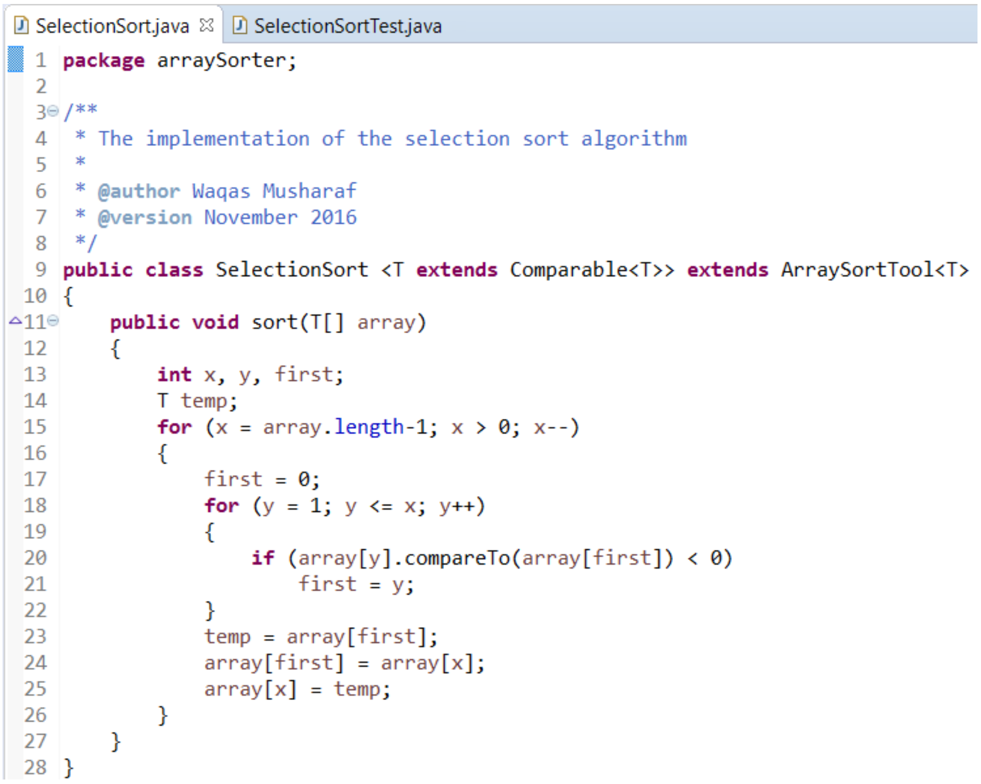
1) Implement the selection sort algorithm. Your implementation should implement the ArraySort interface.

2) Implement the quicksort algorithm. Your implementation should implement the ArraySort interface.

3) Use your implementations to time the execution of these two sorting algorithms for various sizes of array, and plot the results on a graph. Can you arrive at (approximate) formulæ for how the execution times vary in relation to the data size?

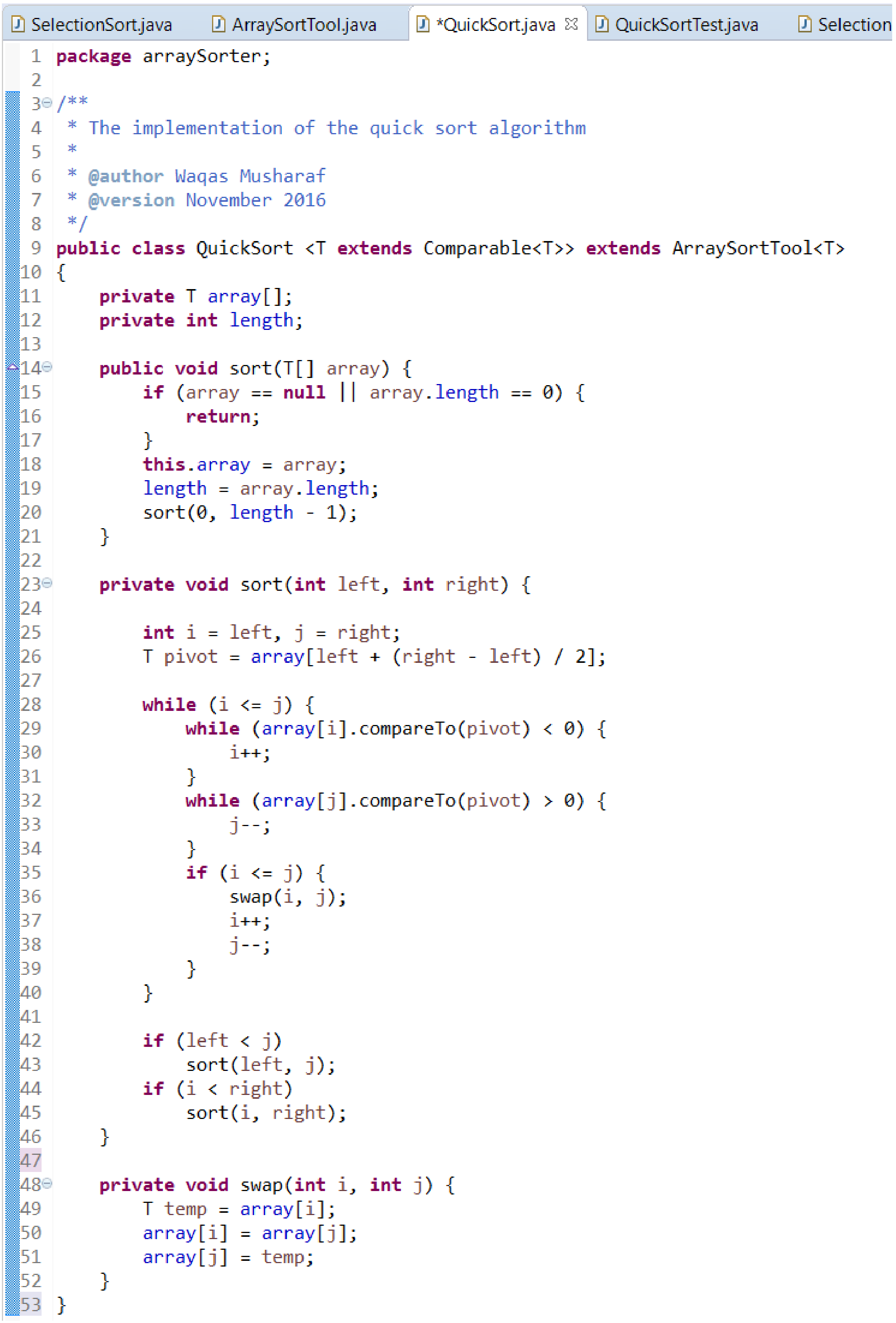
Answers:

1) SelectionSort.java class:



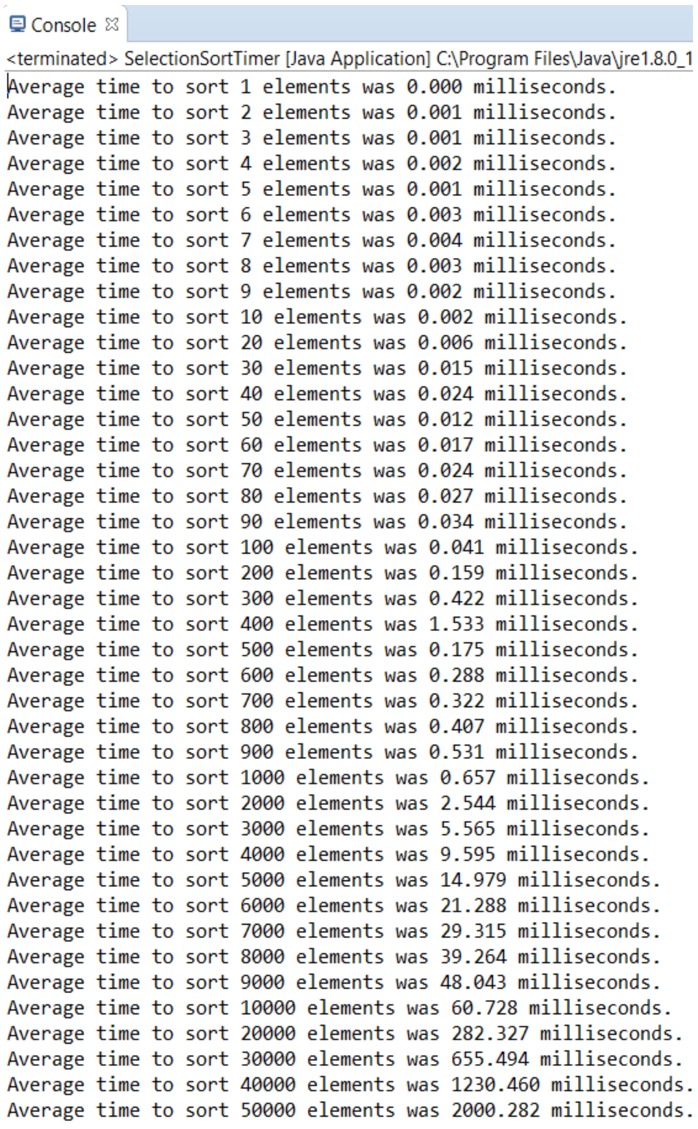
```java
package arraySorter;

/**
 * The implementation of the selection sort algorithm
 *
 * @author Waqas Musharaf
 * @version November 2016
 */
public class SelectionSort <T extends Comparable<T>> extends ArraySortTool<T>
{
    public void sort(T[] array)
    {
        int x, y, first;
        T temp;
        for (x = array.length-1; x > 0; x--)
        {
            first = 0;
            for (y = 1; y <= x; y++)
            {
                if (array[y].compareTo(array[first]) < 0)
                    first = y;
            }
            temp = array[first];
            array[first] = array[x];
            array[x] = temp;
        }
    }
}
```

2) QuickSort.java class:

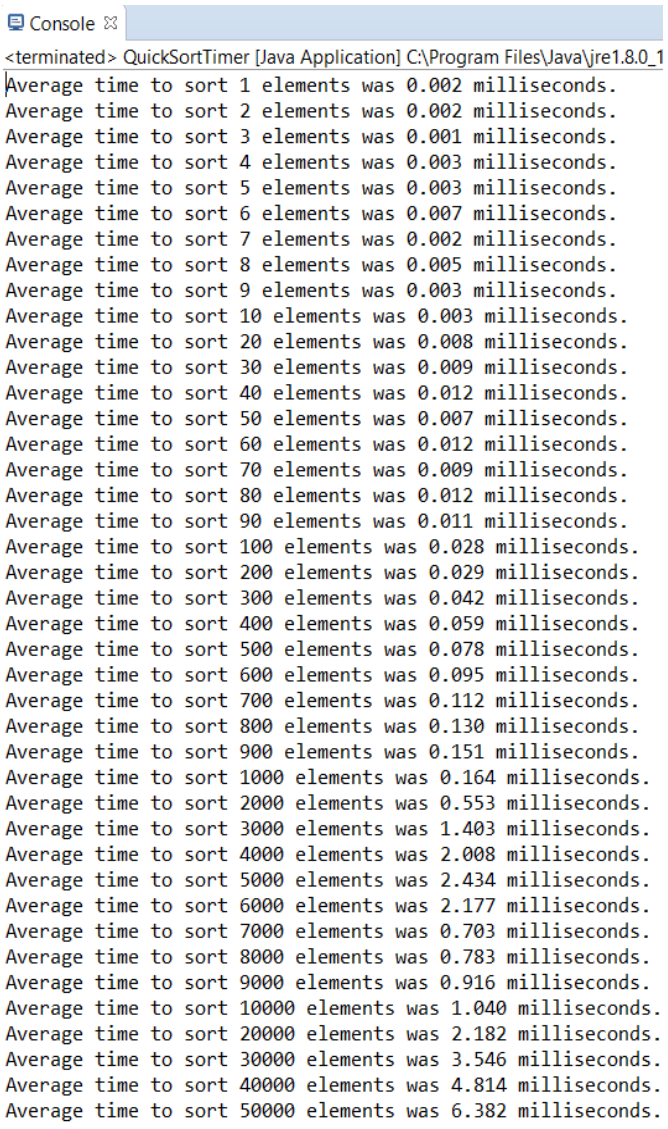


```java
package arraySorter;

/**
 * The implementation of the quick sort algorithm
 *
 * @author Waqas Musharaf
 * @version November 2016
 */
public class QuickSort <T extends Comparable<T>> extends ArraySortTool<T>
{
    private T array[];
    private int length;

    public void sort(T[] array) {
        if (array == null || array.length == 0) {
            return;
        }
        this.array = array;
        length = array.length;
        sort(0, length - 1);
    }

    private void sort(int left, int right) {

        int i = left, j = right;
        T pivot = array[left + (right - left) / 2];

        while (i <= j) {
            while (array[i].compareTo(pivot) < 0) {
                i++;
            }
            while (array[j].compareTo(pivot) > 0) {
                j--;
            }
            if (i <= j) {
                swap(i, j);
                i++;
                j--;
            }
        }

        if (left < j)
            sort(left, j);
        if (i < right)
            sort(i, right);
    }

    private void swap(int i, int j) {
        T temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
}
```

3a) SelectionSortTimer.java console output (left) and QuickSortTimer.java console output (right) for 1-50000 elements:



```
Console ☒
<terminated> SelectionSortTimer [Java Application] C:\Program Files\Java\jre1.8.0_1
Average time to sort 1 elements was 0.000 milliseconds.
Average time to sort 2 elements was 0.001 milliseconds.
Average time to sort 3 elements was 0.001 milliseconds.
Average time to sort 4 elements was 0.002 milliseconds.
Average time to sort 5 elements was 0.001 milliseconds.
Average time to sort 6 elements was 0.003 milliseconds.
Average time to sort 7 elements was 0.004 milliseconds.
Average time to sort 8 elements was 0.003 milliseconds.
Average time to sort 9 elements was 0.002 milliseconds.
Average time to sort 10 elements was 0.002 milliseconds.
Average time to sort 20 elements was 0.006 milliseconds.
Average time to sort 30 elements was 0.015 milliseconds.
Average time to sort 40 elements was 0.024 milliseconds.
Average time to sort 50 elements was 0.012 milliseconds.
Average time to sort 60 elements was 0.017 milliseconds.
Average time to sort 70 elements was 0.024 milliseconds.
Average time to sort 80 elements was 0.027 milliseconds.
Average time to sort 90 elements was 0.034 milliseconds.
Average time to sort 100 elements was 0.041 milliseconds.
Average time to sort 200 elements was 0.159 milliseconds.
Average time to sort 300 elements was 0.422 milliseconds.
Average time to sort 400 elements was 1.533 milliseconds.
Average time to sort 500 elements was 0.175 milliseconds.
Average time to sort 600 elements was 0.288 milliseconds.
Average time to sort 700 elements was 0.322 milliseconds.
Average time to sort 800 elements was 0.407 milliseconds.
Average time to sort 900 elements was 0.531 milliseconds.
Average time to sort 1000 elements was 0.657 milliseconds.
Average time to sort 2000 elements was 2.544 milliseconds.
Average time to sort 3000 elements was 5.565 milliseconds.
Average time to sort 4000 elements was 9.595 milliseconds.
Average time to sort 5000 elements was 14.979 milliseconds.
Average time to sort 6000 elements was 21.288 milliseconds.
Average time to sort 7000 elements was 29.315 milliseconds.
Average time to sort 8000 elements was 39.264 milliseconds.
Average time to sort 9000 elements was 48.043 milliseconds.
Average time to sort 10000 elements was 60.728 milliseconds.
Average time to sort 20000 elements was 282.327 milliseconds.
Average time to sort 30000 elements was 655.494 milliseconds.
Average time to sort 40000 elements was 1230.460 milliseconds.
Average time to sort 50000 elements was 2000.282 milliseconds.
```



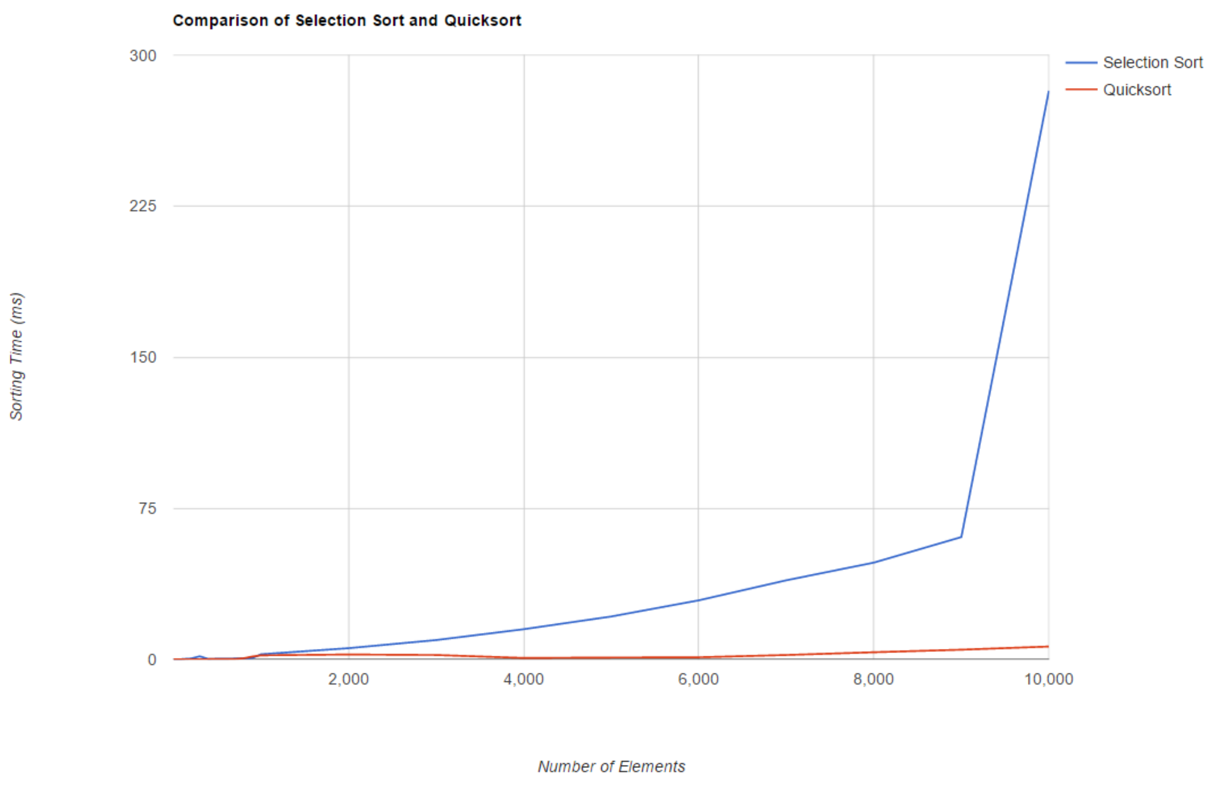
```
Console ☒
<terminated> QuickSortTimer [Java Application] C:\Program Files\Java\jre1.8.0_1
Average time to sort 1 elements was 0.002 milliseconds.
Average time to sort 2 elements was 0.002 milliseconds.
Average time to sort 3 elements was 0.001 milliseconds.
Average time to sort 4 elements was 0.003 milliseconds.
Average time to sort 5 elements was 0.003 milliseconds.
Average time to sort 6 elements was 0.007 milliseconds.
Average time to sort 7 elements was 0.002 milliseconds.
Average time to sort 8 elements was 0.005 milliseconds.
Average time to sort 9 elements was 0.003 milliseconds.
Average time to sort 10 elements was 0.003 milliseconds.
Average time to sort 20 elements was 0.008 milliseconds.
Average time to sort 30 elements was 0.009 milliseconds.
Average time to sort 40 elements was 0.012 milliseconds.
Average time to sort 50 elements was 0.007 milliseconds.
Average time to sort 60 elements was 0.012 milliseconds.
Average time to sort 70 elements was 0.009 milliseconds.
Average time to sort 80 elements was 0.012 milliseconds.
Average time to sort 90 elements was 0.011 milliseconds.
Average time to sort 100 elements was 0.028 milliseconds.
Average time to sort 200 elements was 0.029 milliseconds.
Average time to sort 300 elements was 0.042 milliseconds.
Average time to sort 400 elements was 0.059 milliseconds.
Average time to sort 500 elements was 0.078 milliseconds.
Average time to sort 600 elements was 0.095 milliseconds.
Average time to sort 700 elements was 0.112 milliseconds.
Average time to sort 800 elements was 0.130 milliseconds.
Average time to sort 900 elements was 0.151 milliseconds.
Average time to sort 1000 elements was 0.164 milliseconds.
Average time to sort 2000 elements was 0.553 milliseconds.
Average time to sort 3000 elements was 1.403 milliseconds.
Average time to sort 4000 elements was 2.008 milliseconds.
Average time to sort 5000 elements was 2.434 milliseconds.
Average time to sort 6000 elements was 2.177 milliseconds.
Average time to sort 7000 elements was 0.703 milliseconds.
Average time to sort 8000 elements was 0.783 milliseconds.
Average time to sort 9000 elements was 0.916 milliseconds.
Average time to sort 10000 elements was 1.040 milliseconds.
Average time to sort 20000 elements was 2.182 milliseconds.
Average time to sort 30000 elements was 3.546 milliseconds.
Average time to sort 40000 elements was 4.814 milliseconds.
Average time to sort 50000 elements was 6.382 milliseconds.
```

Selection sort has an average O(n^2) complexity, making it a very inefficient sorting algorithm to sort larger lists, as can be seen in the data above.

Quicksort has an average O(n log n) complexity, resulting in fast and efficient sorts, even with larger lists. This can be seen above, especially in comparison with Selection sort. It is shown that, on average, Quicksort can sort lists of ~40000 elements in approximately the same time Selection sort takes to sort a list of ~3000 elements.

continued on the next page

3b) A graphical comparison of Selection Sort and Quicksort:



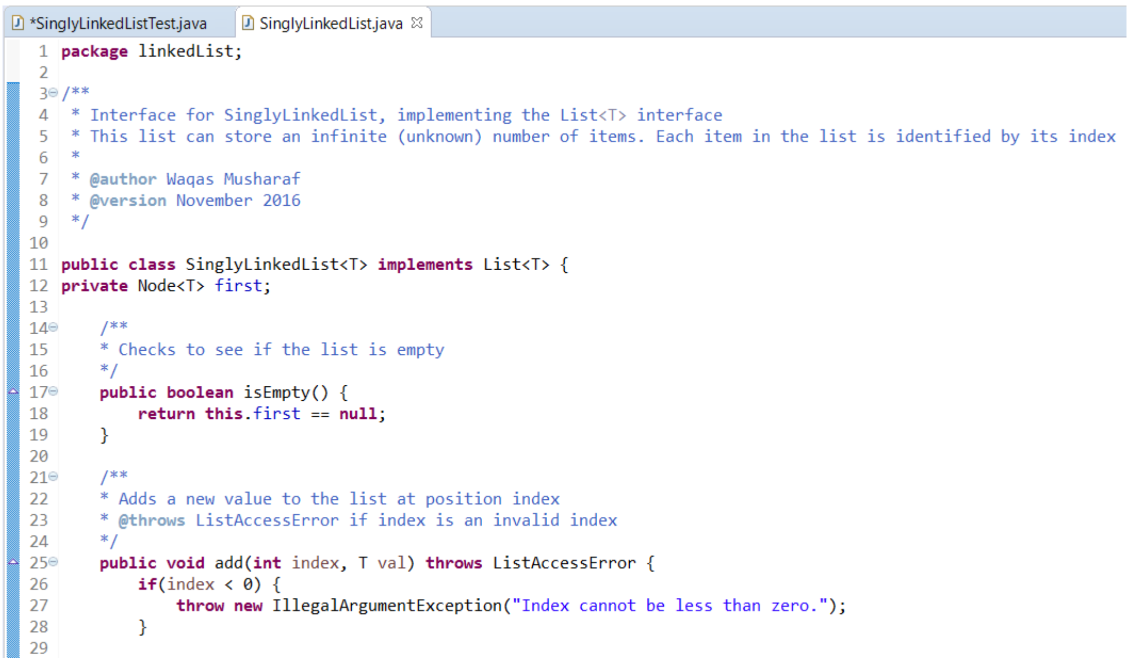
Tutorial Six

Logbook Exercise:

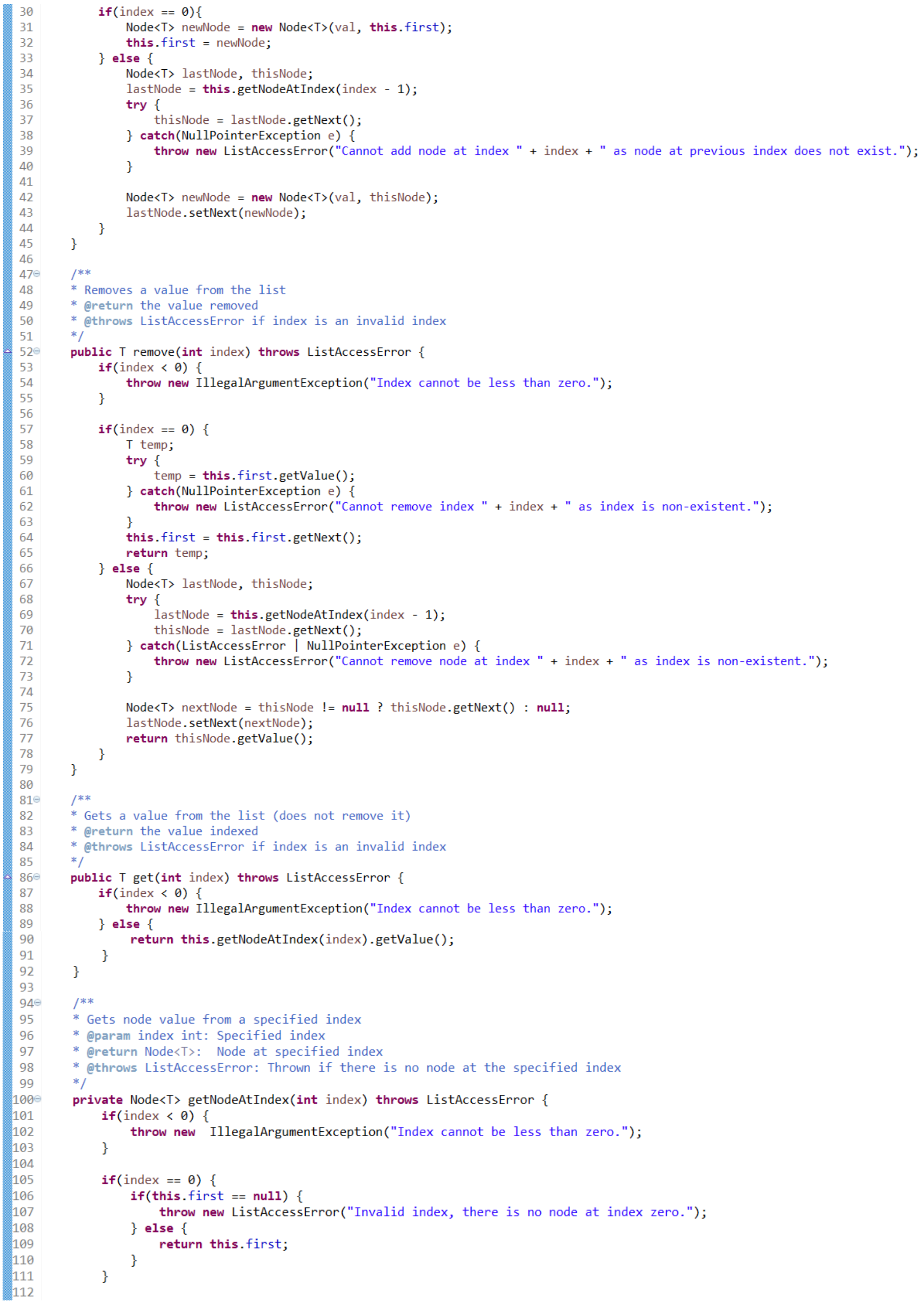
1) Implement the List<T> interface, using singly linked lists.

Answer:

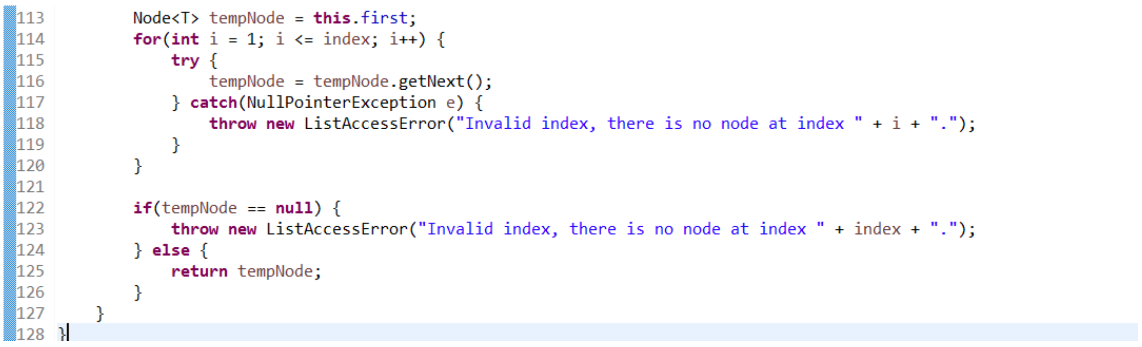
1a) SinglyLinkedList.java class:



```java
package linkedList;

/**
 * Interface for SinglyLinkedList, implementing the List<T> interface
 * This list can store an infinite (unknown) number of items. Each item in the list is identified by its index
 *
 * @author Waqas Musharaf
 * @version November 2016
 */

public class SinglyLinkedList<T> implements List<T> {
private Node<T> first;

    /**
    * Checks to see if the list is empty
    */
    public boolean isEmpty() {
        return this.first == null;
    }

    /**
    * Adds a new value to the list at position index
    * @throws ListAccessError if index is an invalid index
    */
    public void add(int index, T val) throws ListAccessError {
        if(index < 0) {
            throw new IllegalArgumentException("Index cannot be less than zero.");
        }

```

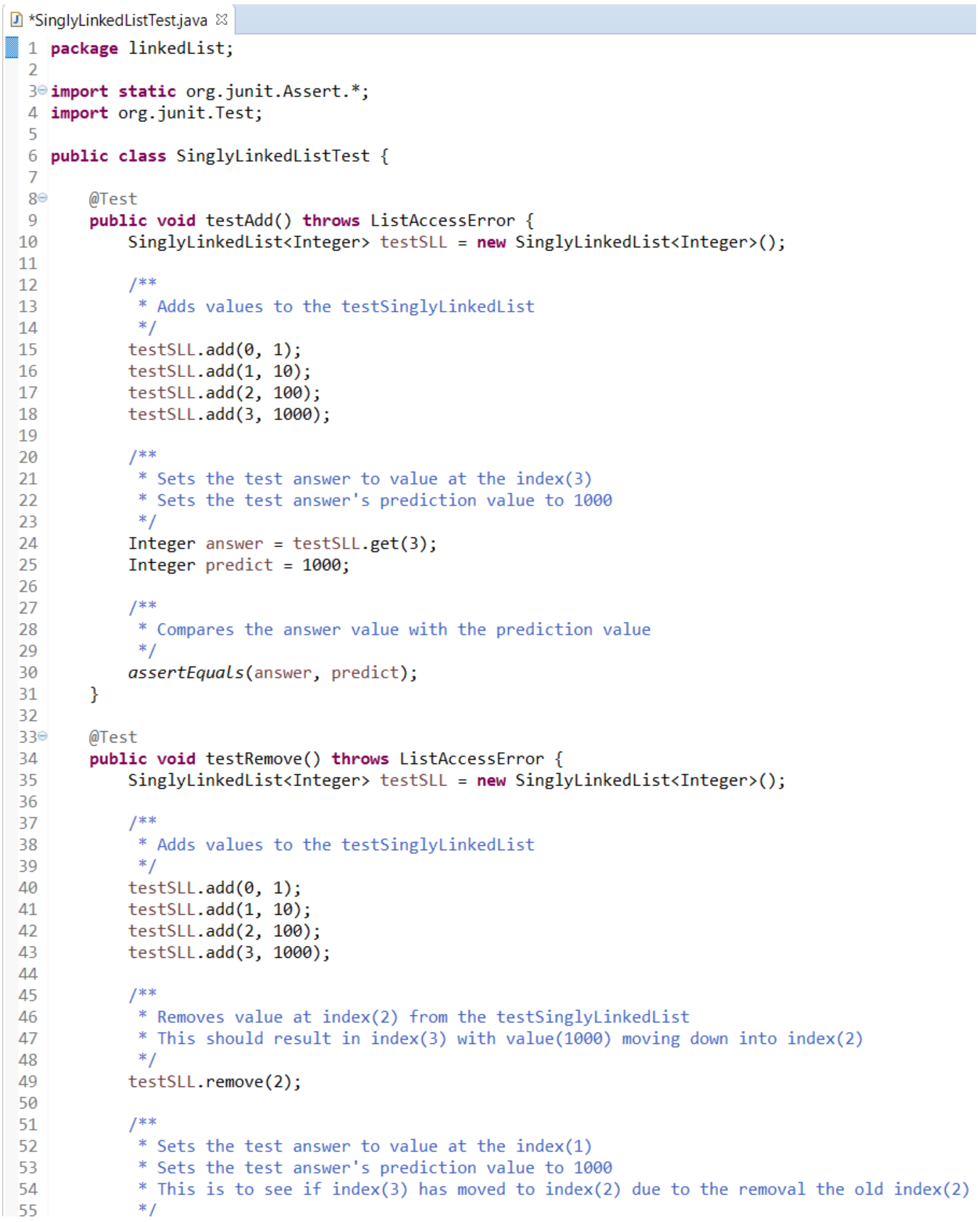


```java
        if(index == 0){
            Node<T> newNode = new Node<T>(val, this.first);
            this.first = newNode;
        } else {
            Node<T> lastNode, thisNode;
            lastNode = this.getNodeAtIndex(index - 1);
            try {
                thisNode = lastNode.getNext();
            } catch(NullPointerException e) {
                throw new ListAccessError("Cannot add node at index " + index + " as node at previous index does not exist.");
            }

            Node<T> newNode = new Node<T>(val, thisNode);
            lastNode.setNext(newNode);
        }
    }

    /**
    * Removes a value from the list
    * @return the value removed
    * @throws ListAccessError if index is an invalid index
    */
    public T remove(int index) throws ListAccessError {
        if(index < 0) {
            throw new IllegalArgumentException("Index cannot be less than zero.");
        }

        if(index == 0) {
            T temp;
            try {
                temp = this.first.getValue();
            } catch(NullPointerException e) {
                throw new ListAccessError("Cannot remove index " + index + " as index is non-existent.");
            }
            this.first = this.first.getNext();
            return temp;
        } else {
            Node<T> lastNode, thisNode;
            try {
                lastNode = this.getNodeAtIndex(index - 1);
                thisNode = lastNode.getNext();
            } catch(ListAccessError | NullPointerException e) {
                throw new ListAccessError("Cannot remove node at index " + index + " as index is non-existent.");
            }

            Node<T> nextNode = thisNode != null ? thisNode.getNext() : null;
            lastNode.setNext(nextNode);
            return thisNode.getValue();
        }
    }

    /**
    * Gets a value from the list (does not remove it)
    * @return the value indexed
    * @throws ListAccessError if index is an invalid index
    */
    public T get(int index) throws ListAccessError {
        if(index < 0) {
            throw new IllegalArgumentException("Index cannot be less than zero.");
        } else {
            return this.getNodeAtIndex(index).getValue();
        }
    }

    /**
    * Gets node value from a specified index
    * @param index int: Specified index
    * @return Node<T>:  Node at specified index
    * @throws ListAccessError: Thrown if there is no node at the specified index
    */
    private Node<T> getNodeAtIndex(int index) throws ListAccessError {
        if(index < 0) {
            throw new  IllegalArgumentException("Index cannot be less than zero.");
        }

        if(index == 0) {
            if(this.first == null) {
                throw new ListAccessError("Invalid index, there is no node at index zero.");
            } else {
                return this.first;
            }
        }

```



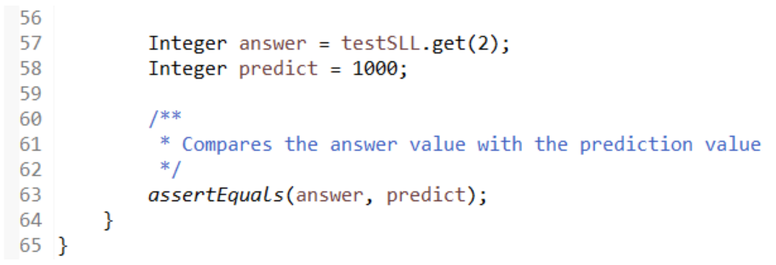
```java
        Node<T> tempNode = this.first;
        for(int i = 1; i <= index; i++) {
            try {
                tempNode = tempNode.getNext();
            } catch(NullPointerException e) {
                throw new ListAccessError("Invalid index, there is no node at index " + i + ".");
            }
        }

        if(tempNode == null) {
            throw new ListAccessError("Invalid index, there is no node at index " + index + ".");
        } else {
            return tempNode;
        }
    }
}
```

1b) JUnit tests for SinglyLinkedList.java class, using SinglyLinkedListTest.java:

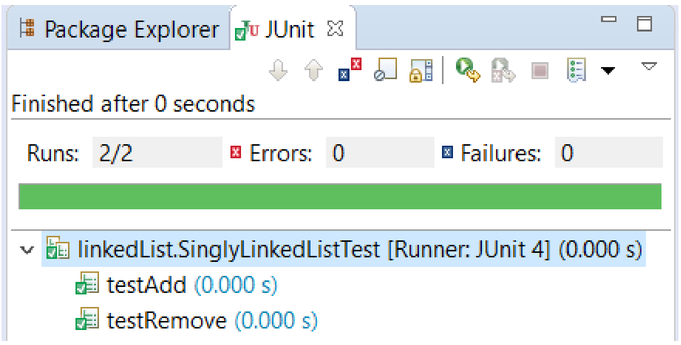


*SinglyLinkedListTest.java

```java
package linkedList;

import static org.junit.Assert.*;
import org.junit.Test;

public class SinglyLinkedListTest {

    @Test
    public void testAdd() throws ListAccessError {
        SinglyLinkedList<Integer> testSLL = new SinglyLinkedList<Integer>();

        /**
         * Adds values to the testSinglyLinkedList
         */
        testSLL.add(0, 1);
        testSLL.add(1, 10);
        testSLL.add(2, 100);
        testSLL.add(3, 1000);

        /**
         * Sets the test answer to value at the index(3)
         * Sets the test answer's prediction value to 1000
         */
        Integer answer = testSLL.get(3);
        Integer predict = 1000;

        /**
         * Compares the answer value with the prediction value
         */
        assertEquals(answer, predict);
    }

    @Test
    public void testRemove() throws ListAccessError {
        SinglyLinkedList<Integer> testSLL = new SinglyLinkedList<Integer>();

        /**
         * Adds values to the testSinglyLinkedList
         */
        testSLL.add(0, 1);
        testSLL.add(1, 10);
        testSLL.add(2, 100);
        testSLL.add(3, 1000);

        /**
         * Removes value at index(2) from the testSinglyLinkedList
         * This should result in index(3) with value(1000) moving down into index(2)
         */
        testSLL.remove(2);

        /**
         * Sets the test answer to value at the index(1)
         * Sets the test answer's prediction value to 1000
         * This is to see if index(3) has moved to index(2) due to the removal the old index(2)
         */
```



```

        Integer answer = testSLL.get(2);
        Integer predict = 1000;

        /**
         * Compares the answer value with the prediction value
         */
        assertEquals(answer, predict);
    }
}
```



```
Finished after 0 seconds

Runs: 2/2    Errors: 0    Failures: 0

linkedList.SinglyLinkedListTest [Runner: JUnit 4] (0.000 s)
    testAdd (0.000 s)
    testRemove (0.000 s)
```

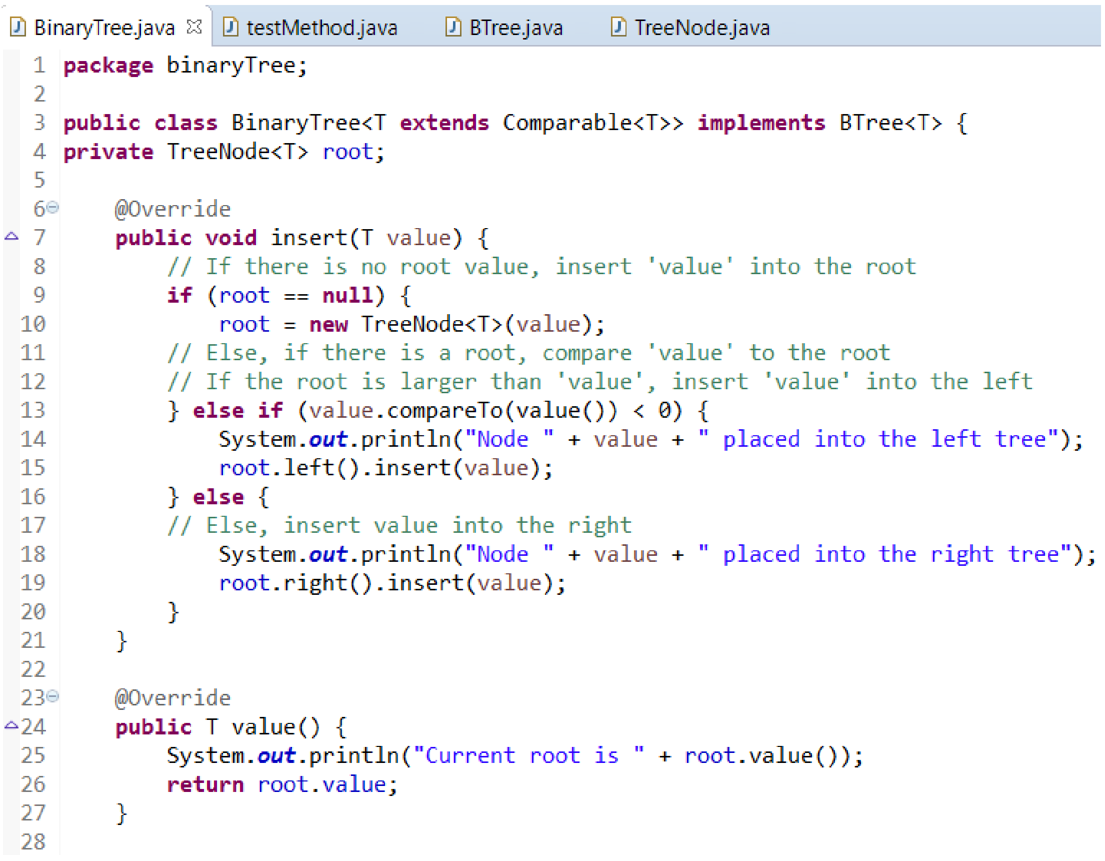
Tutorial Seven

Logbook Exercise:

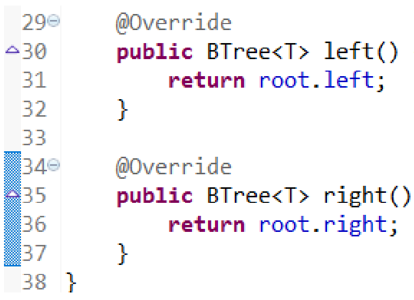
1) Complete the implementation of the binary tree class shown in the lecture (no code is provided — you should build up this class from scratch).

Answer:

1a) BinaryTree.java class:

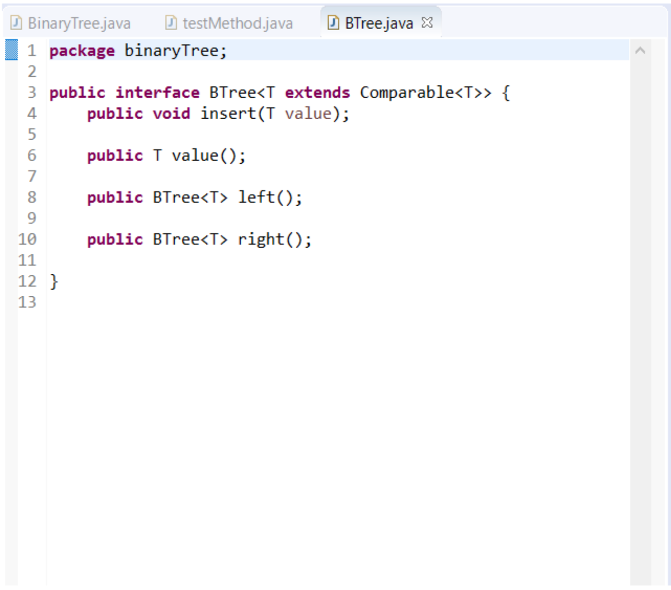


```
package binaryTree;

public class BinaryTree<T extends Comparable<T>> implements BTree<T> {
private TreeNode<T> root;

    @Override
    public void insert(T value) {
        // If there is no root value, insert 'value' into the root
        if (root == null) {
            root = new TreeNode<T>(value);
        // Else, if there is a root, compare 'value' to the root
        // If the root is larger than 'value', insert 'value' into the left
        } else if (value.compareTo(value()) < 0) {
            System.out.println("Node " + value + " placed into the left tree");
            root.left().insert(value);
        } else {
        // Else, insert value into the right
            System.out.println("Node " + value + " placed into the right tree");
            root.right().insert(value);
        }
    }

    @Override
    public T value() {
        System.out.println("Current root is " + root.value());
        return root.value;
    }

```

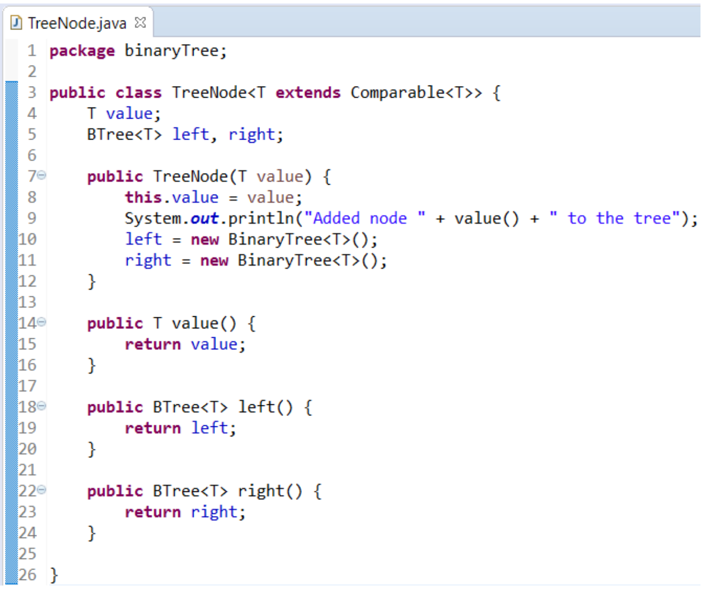


```java
    @Override
    public BTree<T> left() {
        return root.left;
    }

    @Override
    public BTree<T> right() {
        return root.right;
    }
}
```

1b) BTree.java and TreeNode.java classes:

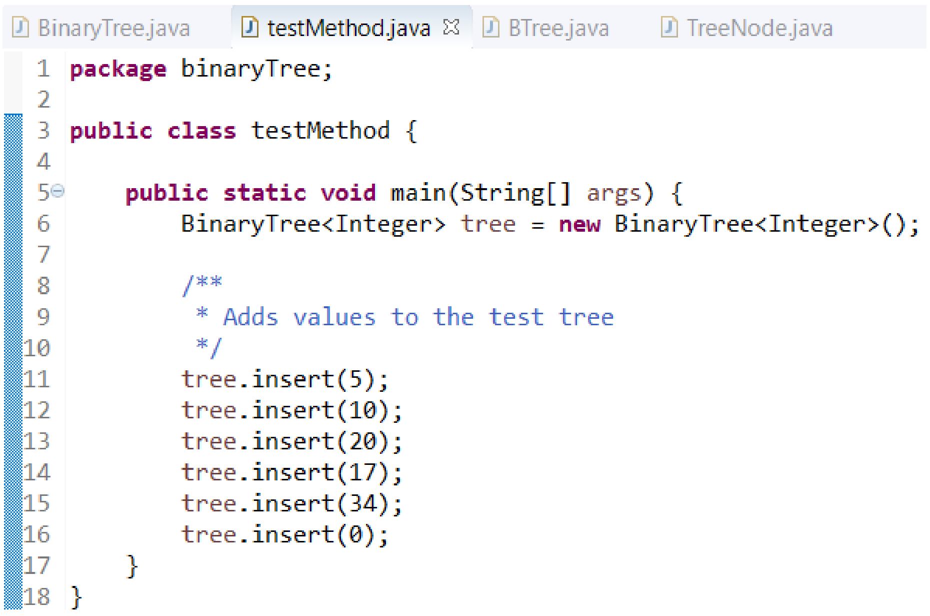


```java
package binaryTree;

public interface BTree<T extends Comparable<T>> {
    public void insert(T value);

    public T value();

    public BTree<T> left();

    public BTree<T> right();
}
```



```java
package binaryTree;

public class TreeNode<T extends Comparable<T>> {
    T value;
    BTree<T> left, right;

    public TreeNode(T value) {
        this.value = value;
        System.out.println("Added node " + value() + " to the tree");
        left = new BinaryTree<T>();
        right = new BinaryTree<T>();
    }

    public T value() {
        return value;
    }

    public BTree<T> left() {
        return left;
    }

    public BTree<T> right() {
        return right;
    }
}
```

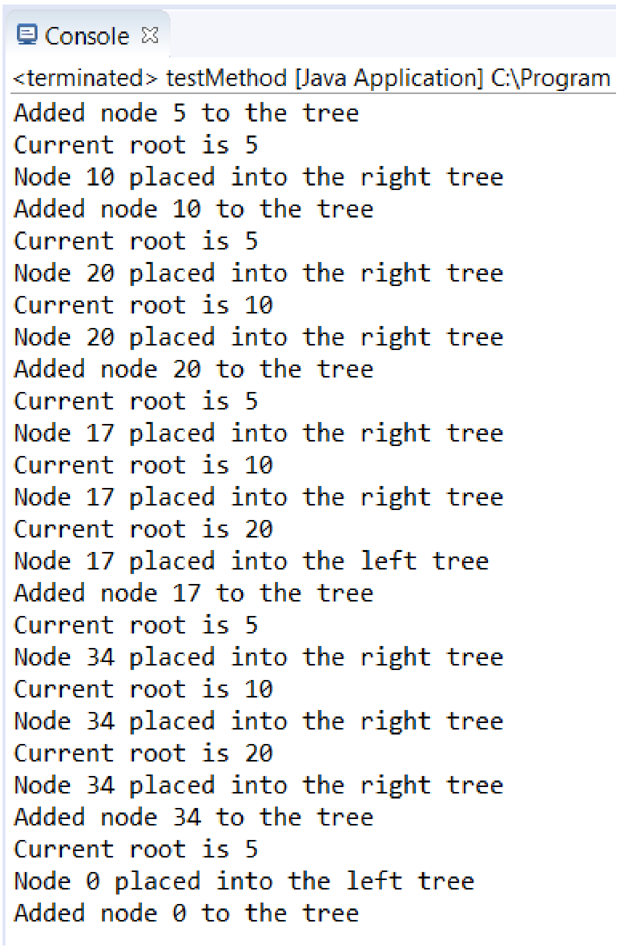
1c) testMethod.java class:



```java
package binaryTree;

public class testMethod {

    public static void main(String[] args) {
        BinaryTree<Integer> tree = new BinaryTree<Integer>();

        /**
         * Adds values to the test tree
         */
        tree.insert(5);
        tree.insert(10);
        tree.insert(20);
        tree.insert(17);
        tree.insert(34);
        tree.insert(0);
    }
}
```

continued on the next page

1d) testMethod.java console output:



```
Console ⊠
<terminated> testMethod [Java Application] C:\Program
Added node 5 to the tree
Current root is 5
Node 10 placed into the right tree
Added node 10 to the tree
Current root is 5
Node 20 placed into the right tree
Current root is 10
Node 20 placed into the right tree
Added node 20 to the tree
Current root is 5
Node 17 placed into the right tree
Current root is 10
Node 17 placed into the right tree
Current root is 20
Node 17 placed into the left tree
Added node 17 to the tree
Current root is 5
Node 34 placed into the right tree
Current root is 10
Node 34 placed into the right tree
Current root is 20
Node 34 placed into the right tree
Added node 34 to the tree
Current root is 5
Node 0 placed into the left tree
Added node 0 to the tree
```

Due to Week 8 being a Guidance Week, there were no lecture or tutorial sessions and therefore no logbook exercises to complete. There was, however, a self-assessment form completed during Week 8, which can be seen below:

Self-Assessment One

This self-assessment was completed during Week 8, Guidance Week. Some work or comments may have changed since the time of writing.

Week(s)	Overall	Documentation	Structure	Naming	Testing	Functionality
1 & 2 Search Timer	A	A	A+	A+	B	B
3 & 4 Generic Swap	A+	A+	A+	A	A	A+
5 Sorting	A+	A+	A+	A+	A	A+
6 Linked Lists	A	A	A+	A	B	A
7 Binary Trees	A	A	A+	A+	B	A

Assessment Criterion	Grade
Answers to flagged logbook questions	**A**
Answers to other practical questions	**C**
Other practical work	**C**
Understanding of the module material to date	**A**
Level of self-reflection & evaluation	**A+**
Participation in timetabled activities	**A**
Time spent outside timetabled classes	**B**

Comments:

Weeks 1 & 2: I believe that the quality of documentation for these exercises was good. It could have been further improved with additional comments in the extracted code, however, I didn't feel that the documentation was lacking without them. Structure and naming is, in my opinion, strong throughout the code. Some testing was lacking in these exercises though, and I plan to add some more testing later. The functionality of the code seems strong, but cannot be truly attested without more testing documentation.

Weeks 3 & 4: I believe that the quality of documentation for these exercises was high, with sufficient comments throughout the code. I also believe that the structure and naming were of a very high standard and very easy to read. Testing is sufficient, as a test method is documented with expected values and actual values shown. I would say that the code for these exercises was of high functionality.

Week 5: I believe that the quality of the documentation for this exercise was strong, as it includes screenshots of all code, timings and graphs necessary. I also believe that the structure and naming were of a very high standard and very easy to read. Testing was also strong, as the timings for both sorts were displayed and graphed. I would say that the code for this exercise was of high functionality.

Week 6: I believe that the quality of documentation for this exercise was satisfactory. It could have been improved with additional comments in the extracted code, however, I didn't feel that the documentation was lacking without them. The structure and naming was of a high standard throughout. However, some testing was lacking in this exercise, and could be something I could improve upon soon. The functionality of the code seems strong, according to the tests taken and documented so far

Week 7: I believe that the quality of documentation for this exercise was satisfactory. It could have been improved with additional comments in the extracted code, however, I didn't feel that the documentation was lacking without them. Structure and naming is, in my opinion, strong throughout the code. Some testing was lacking in this exercise though, and I plan to add some further testing later. The functionality of the code seems strong, according to the tests taken and documented so far.

Overall, I believe that I have answered the flagged logbook questions strongly, although additional work has been lacking. I should strive to add more additional work to the logbook as I progress with the module. So far, my understanding of this module's material is good, but could be enhanced further with additional reading. I believe my engagement with the module during lessons is also good, as I attend most of my timetabled sessions, or other sessions that I can attend if I need to. An area to improve on may be the time I spend on this module outside of lessons, and I should aim to plan my time better to be able to spend more time on this module.

Tutorial Nine

Logbook Exercise:

1) Create an object instance of the HashtableWrapper <String, Integer> class (this is essentially Java's standard java.util.Hashtable class). Ensure this hashtable has size 5.

- Inspect the object you have just created, paying attention to the object's internal array.

*- Now, using the **void** put (String key, Integer data) method, inherited from Hashtable, add the key/value pair ("fred",37) to the hashtable ("fred" is the key, 37 is the value). Inspect the object again.*

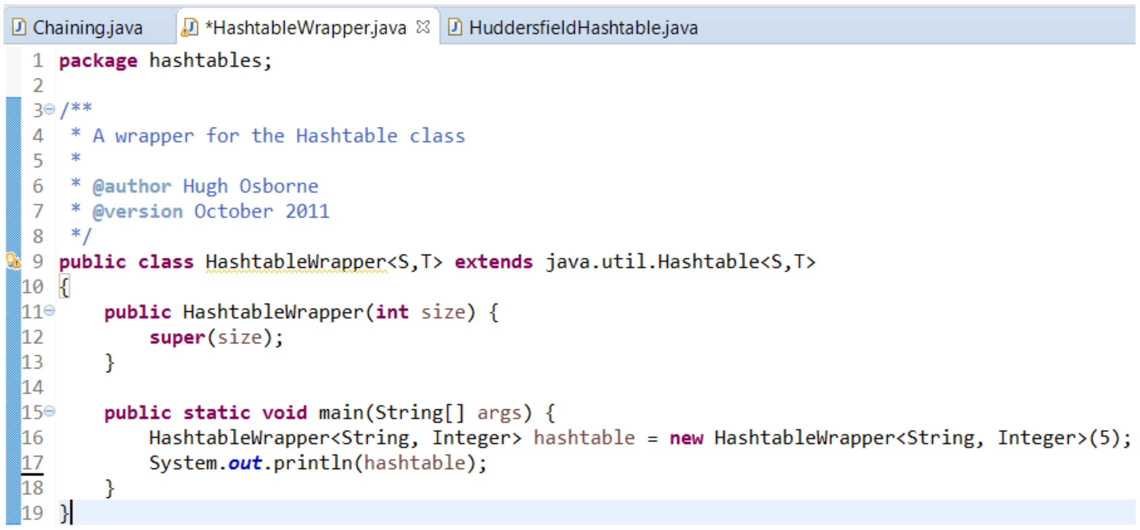
- Now add the following key/value pairs, again inspecting the hashtable object after each new pair is entered: ("is",69), ("dead",0), ("but",999), ("not", -42), ("me!", -1)

- Describe, and explain what happens.

Answer:

1)

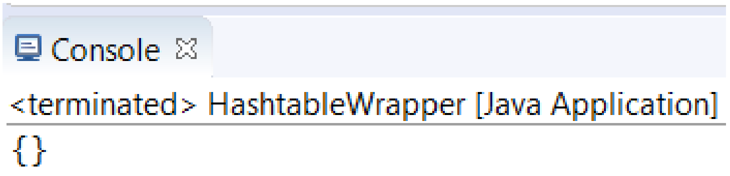
The screenshot below shows the creation of an object instance of the *HashtableWrapper <String, Integer>* class. The size of the hashtable object has been specified as 5:



Chaining.java | *HashtableWrapper.java | HuddersfieldHashtable.java

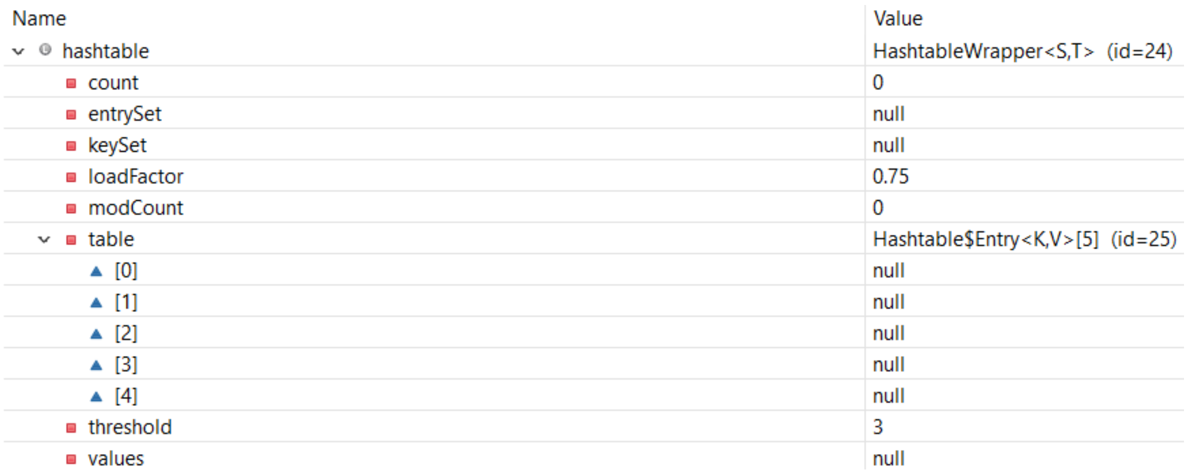
```java
package hashtables;

/**
 * A wrapper for the Hashtable class
 *
 * @author Hugh Osborne
 * @version October 2011
 */
public class HashtableWrapper<S,T> extends java.util.Hashtable<S,T>
{
    public HashtableWrapper(int size) {
        super(size);
    }

    public static void main(String[] args) {
        HashtableWrapper<String, Integer> hashtable = new HashtableWrapper<String, Integer>(5);
        System.out.println(hashtable);
    }
}
```

The screenshot below shows the console upon running the updated main method containing the *HashtableWrapper <String, Integer>* object. The console shows that the hashtable (represented as '{}') is initially empty as it contains no elements:



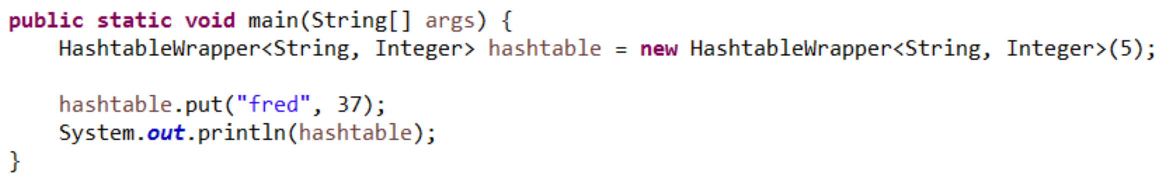
```
<terminated> HashtableWrapper [Java Application]
{}
```

However, using the Variables view in Debug mode of the Eclipse IDE, allows the newly created hashtable to be inspected in greater detail: This view displays various attributes of the hashtable object created. This includes confirming that the hashtable does indeed have an internal array of 5 fields, all of which are currently null (empty), and the loadFactor of the hashtable (which will be covered in more detail later).



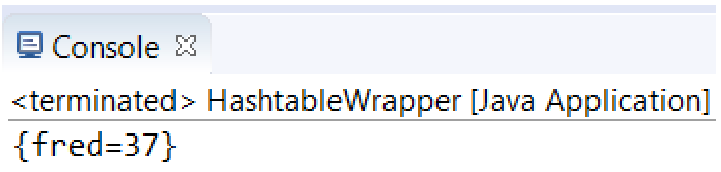
Name	Value
hashtable	HashtableWrapper<S,T> (id=24)
count	0
entrySet	null
keySet	null
loadFactor	0.75
modCount	0
table	Hashtable$Entry<K,V>[5] (id=25)
[0]	null
[1]	null
[2]	null
[3]	null
[4]	null
threshold	3
values	null

Next, this screenshot shows an updated main method using the void ‘put’ method inherited from Hashtable and a key/value pair of “fred”/37:



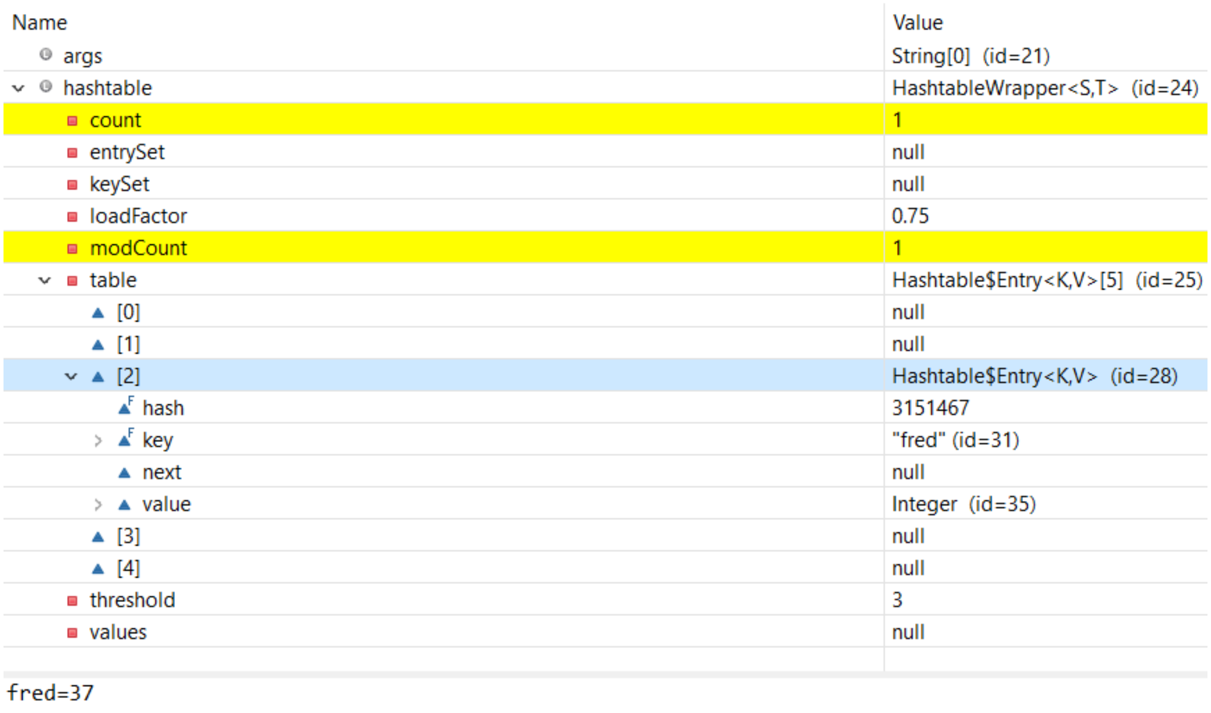
```
public static void main(String[] args) {
    HashtableWrapper<String, Integer> hashtable = new HashtableWrapper<String, Integer>(5);

    hashtable.put("fred", 37);
    System.out.println(hashtable);
}
```

Here is the console output upon running the updated *HashtableWrapper.java:* It can be observed that the hashtable now shows the key specified (“fred”) as mapped to the value specified (37) in the form ‘key=value’.



```
Console
<terminated> HashtableWrapper [Java Application]
{fred=37}
```

The Variables view allows us to view the hashtable in more detail: It can be observed that the count and the modCount of the hashtable have both increased by one, this is due to the addition of the first key/value pair into the hashtable. The key/value pair “fred”/37 can be seen to have been inserted into the index[2] of the hashtable. This is due to the hashtable hash function. The hash of “fred”/37 can be seen to be 3151467, which is a value generated from the key (in this case “fred”). The hash function takes this hash value and performs the modulo of the current array size onto the hash, which in this case would be 3151467-modulo-5. The result of this hash function (in this case 2) is the index into which the key/value pair is inserted.



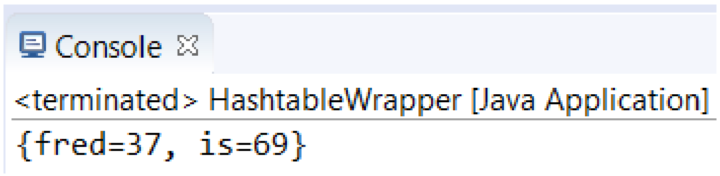
Name	Value
args	String[0] (id=21)
hashtable	HashtableWrapper<S,T> (id=24)
count	1
entrySet	null
keySet	null
loadFactor	0.75
modCount	1
table	Hashtable$Entry<K,V>[5] (id=25)
[0]	null
[1]	null
[2]	Hashtable$Entry<K,V> (id=28)
hash	3151467
key	"fred" (id=31)
next	null
value	Integer (id=35)
[3]	null
[4]	null
threshold	3
values	null

```
fred=37
```

Next, this screenshot shows a further updated main method, again, using the void 'put' method inherited from Hashtable and an additional key/value pair of "is"/69:

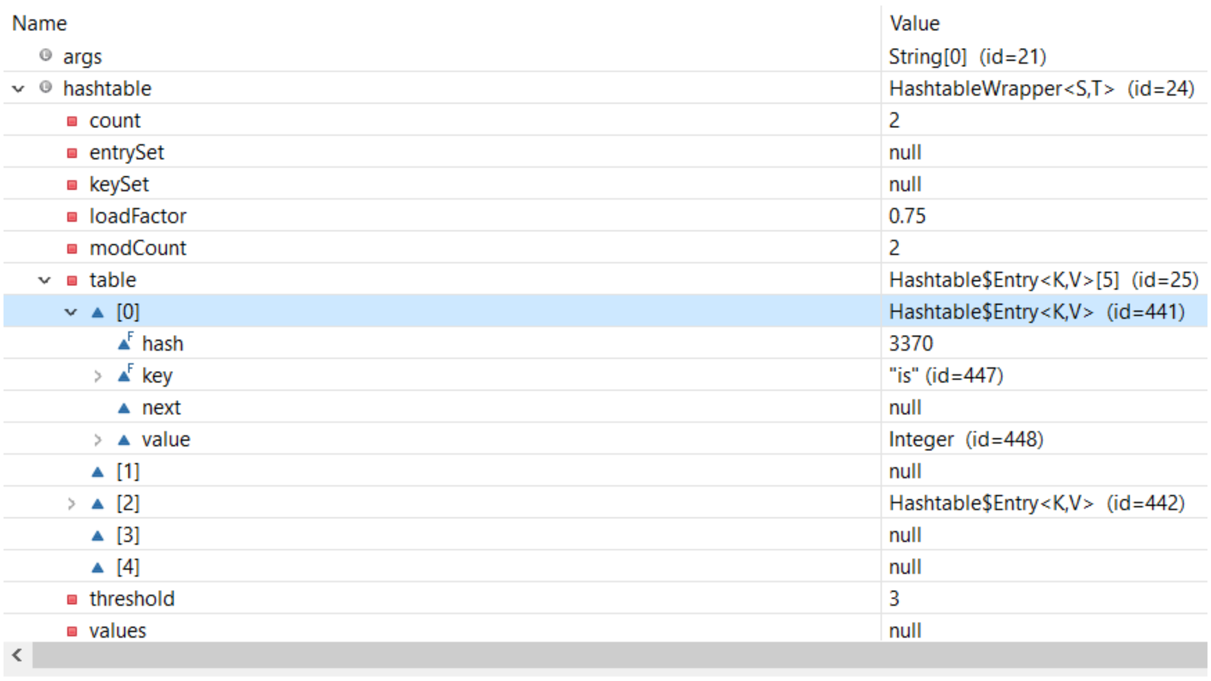
```
public static void main(String[] args) {
    HashtableWrapper<String, Integer> hashtable = new HashtableWrapper<String, Integer>(5);

    hashtable.put("fred", 37);
    hashtable.put("is", 69);
    System.out.println(hashtable);
}
```

Here is the console output upon running the updated *HashtableWrapper.java:* It can be observed that the hashtable now contains two key/value pairs; "fred"/37 followed by "is"/69. The key/value pairs are shown in the form 'key=value' and each pair is separated by a comma.



```
Console
<terminated> HashtableWrapper [Java Application]
{fred=37, is=69}
```

The Variables view allows to see the hashtable in more detail: It can be observed that the count and the modCount of the hashtable have again both increased by one, with the addition of the second key/value pair into the hashtable. The new key/value pair has been inserted into index[0]. This is because the hash of "is"/69 can be seen to be 3370, which has a 5th modulus of zero, resulting in "is"/69 being placed in index[0].



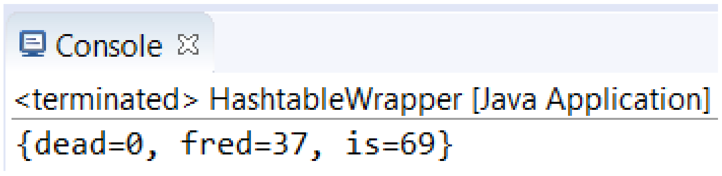
Name	Value
args	String[0] (id=21)
hashtable	HashtableWrapper<S,T> (id=24)
count	2
entrySet	null
keySet	null
loadFactor	0.75
modCount	2
table	Hashtable$Entry<K,V>[5] (id=25)
[0]	Hashtable$Entry<K,V> (id=441)
hash	3370
key	"is" (id=447)
next	null
value	Integer (id=448)
[1]	null
[2]	Hashtable$Entry<K,V> (id=442)
[3]	null
[4]	null
threshold	3
values	null

```
is=69
```

Next, this screenshot shows a further updated main method, again, using the void ‘put’ method inherited from Hashtable and an additional key/value pair of “dead”/0:

```
    public static void main(String[] args) {
        HashtableWrapper<String, Integer> hashtable = new HashtableWrapper<String, Integer>(5);

        hashtable.put("fred", 37);
        hashtable.put("is", 69);
        hashtable.put("dead", 0);
        System.out.println(hashtable);
    }
```

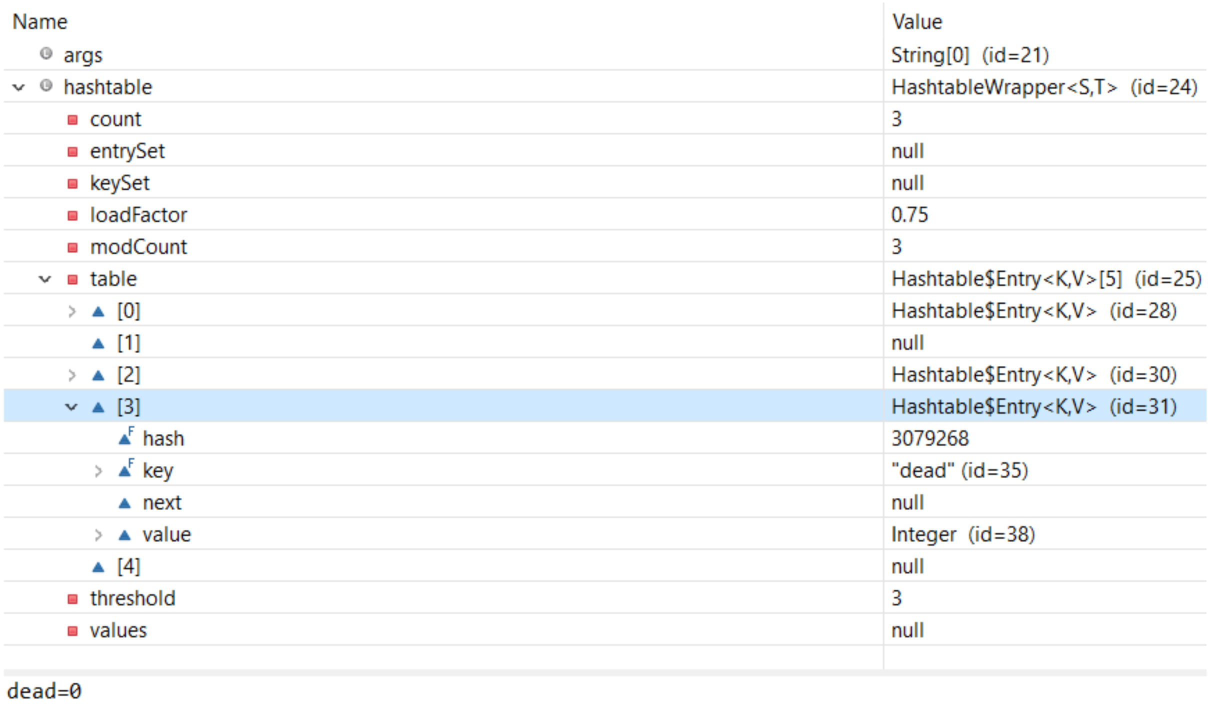
Here is the console output upon running the updated *HashtableWrapper.java:* It can be observed that the hashtable now contains an additional key/value pair.



Console

```
<terminated> HashtableWrapper [Java Application]
{dead=0, fred=37, is=69}
```

Upon inspecting the Variables view of the hashtable, it be observed that the count and the modCount of the hashtable have again both increased by one, with the addition of the third key/value pair into the hashtable. The new key/value pair has been inserted into index[3]. This is because the hash of “dead”/0 can be seen to be 3079268, which has a 5th modulus of 3, resulting in “dead”/0 being placed in index[3].



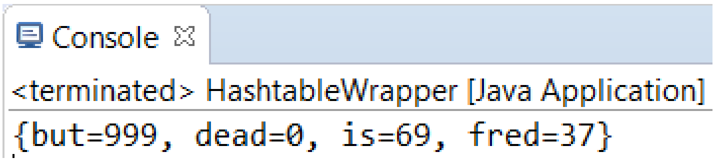
Name	Value
args	String[0] (id=21)
hashtable	HashtableWrapper<S,T> (id=24)
count	3
entrySet	null
keySet	null
loadFactor	0.75
modCount	3
table	Hashtable$Entry<K,V>[5] (id=25)
[0]	Hashtable$Entry<K,V> (id=28)
[1]	null
[2]	Hashtable$Entry<K,V> (id=30)
[3]	Hashtable$Entry<K,V> (id=31)
hash	3079268
key	"dead" (id=35)
next	null
value	Integer (id=38)
[4]	null
threshold	3
values	null

```
dead=0
```

Next, this screenshot shows a further updated main method, again, using the void 'put' method inherited from Hashtable and an additional key/value pair of "but"/999:

```
public static void main(String[] args) {
    HashtableWrapper<String, Integer> hashtable = new HashtableWrapper<String, Integer>(5);

    hashtable.put("fred", 37);
    hashtable.put("is", 69);
    hashtable.put("dead", 0);
    hashtable.put("but", 999);
    System.out.println(hashtable);
}
```

Here is the console output upon running the updated *HashtableWrapper.java:* It can be observed that the hashtable now contains an additional key/value pair.



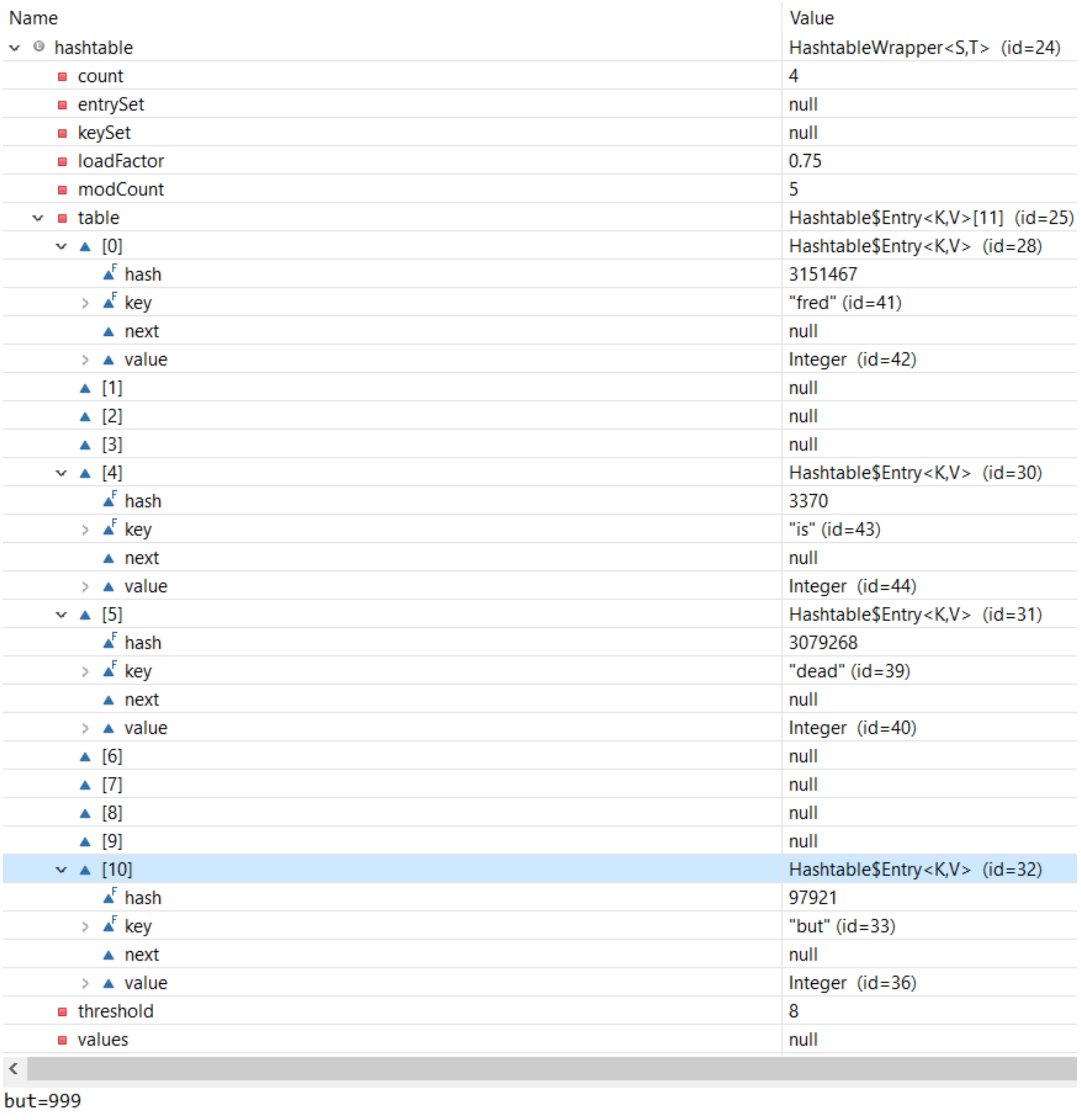
```
Console
<terminated> HashtableWrapper [Java Application]
{but=999, dead=0, is=69, fred=37}
```

Upon inspecting the Variables view of the hashtable, it be observed that the count of the hashtable has increased by one, but the modCount has increased by two, with the addition of the fourth key/value pair into the hashtable. The threshold value has also increased from 3 to 8.

A more obvious change to the hashtable, however, is that the size of the hashtable has increased from 5 to 11. This change in hashtable size is due to two things; the amount of data in the hashtable and the loadValue. The hashtable originally had a size of 5, but when the 4th key/value pair was inserted, the hashtable capacity exceeded the loadValue. The loadValue for this hashtable is 0.75, which means to exceed the load value, 75% of the hashtable must be full. When the 4th key/value pair was inserted, the capacity of the hashtable increased from 60% to 80%, which is above the load value. The threshold value is another way of looking at this; the threshold value was previously 3, which was the maximum capacity of the hashtable without exceeding the loadValue. When the loadValue is exceeded, the hashtable automatically increases in size to accommodate future data. The new size of the hashtable is determined by the formula 2n+1, where 'n' is the previous size of the hashtable. Therefore, the hashtable was resized to (2*5) + 1, which is equal to 11.

With the size of the hashtable now increased, the positioning of key/value pairs within in the hashtable has also changed. This is because the previous positions were calculated with a modulus of 5, due to the previous size of the hashtable; the current positions are calculated with a new modulus, modulus 11. This has resulted in key/value pair "fred"/37 moving to index[0], "is/69" moving to index[4], "dead"/0 moving to index[5] and the new key/value pair of "but/999" being inserted into index[10]

The Variables view of the current hashtable object can be seen below:



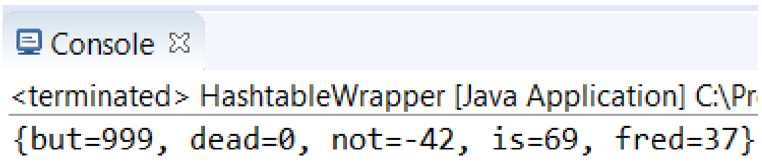
Name	Value
hashtable	HashtableWrapper<S,T> (id=24)
count	4
entrySet	null
keySet	null
loadFactor	0.75
modCount	5
table	Hashtable$Entry<K,V>[11] (id=25)
[0]	Hashtable$Entry<K,V> (id=28)
hash	3151467
key	"fred" (id=41)
next	null
value	Integer (id=42)
[1]	null
[2]	null
[3]	null
[4]	Hashtable$Entry<K,V> (id=30)
hash	3370
key	"is" (id=43)
next	null
value	Integer (id=44)
[5]	Hashtable$Entry<K,V> (id=31)
hash	3079268
key	"dead" (id=39)
next	null
value	Integer (id=40)
[6]	null
[7]	null
[8]	null
[9]	null
[10]	Hashtable$Entry<K,V> (id=32)
hash	97921
key	"but" (id=33)
next	null
value	Integer (id=36)
threshold	8
values	null

```
but=999
```

Next, this screenshot shows a further updated main method, again, using the void 'put' method inherited from Hashtable and an additional key/value pair of "not"/-42:

```
    public static void main(String[] args) {
        HashtableWrapper<String, Integer> hashtable = new HashtableWrapper<String, Integer>(5);

        hashtable.put("fred", 37);
        hashtable.put("is", 69);
        hashtable.put("dead", 0);
        hashtable.put("but", 999);
        hashtable.put("not", -42);
        System.out.println(hashtable);
    }
```

Here is the console output upon running the updated *HashtableWrapper.java:* It can be observed that the hashtable now contains an additional key/value pair.



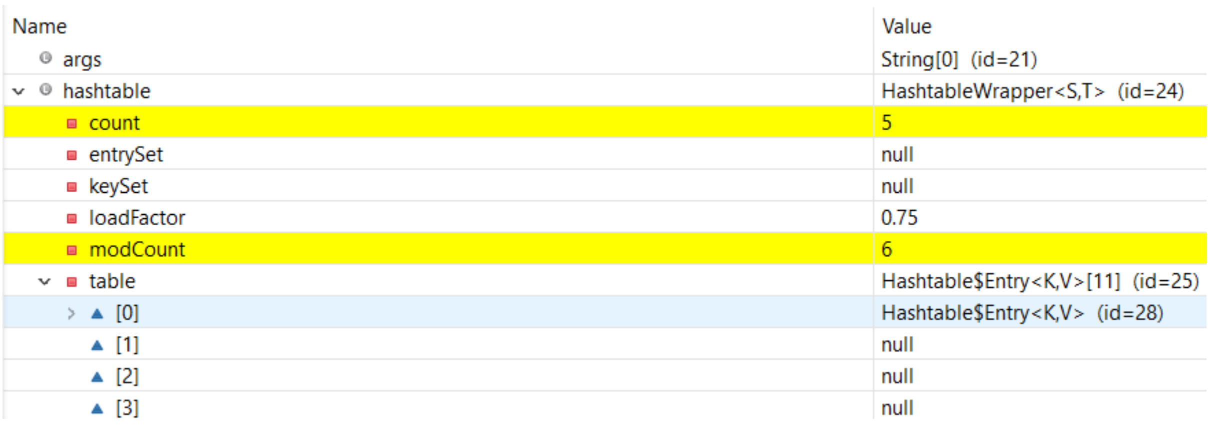
```
Console
<terminated> HashtableWrapper [Java Application] C:\Pr
{but=999, dead=0, not=-42, is=69, fred=37}
```

Upon inspecting the Variables view of the hashtable, it be observed that the count and the modCount of the hashtable have again both increased by one, with the addition of the fifth key/value pair into the hashtable. However, upon closer inspection, none of the previously null indexes are now occupied, and the same number of occupied indexes remain as before this insertion.

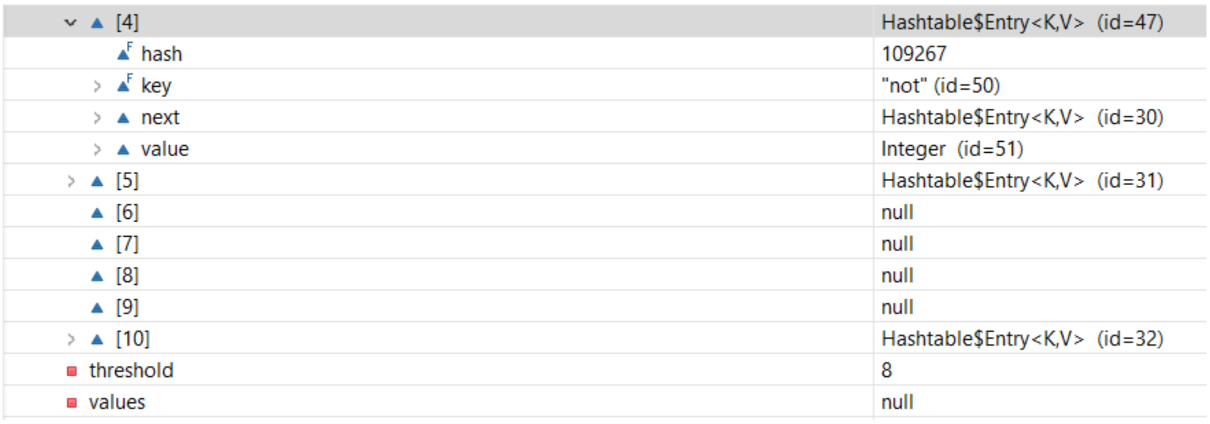
This is because "not"/-42 has a hash value of 109267, which has an 11th modulus of 4. This results in "not"/-42 being placed in index[4]. However, there was previously already a key/value pair of "is"/69 at index[4] which seems to have been collided with the new key/value pair. This is called address collision. In event of such collisions, there are memory management approaches that deal with the collision. Open addressing (explicit memory management) deals with collisions by finding another unoccupied data location, whilst chaining (implicit memory management) simply adds the colliding data to the colliding position's linked list (as with chaining, data is stored in a linked list).

Since the same number of indexes are occupied as before the addition of "not"/-42 and therefore, another unoccupied data location was not found to store the data, the memory management approach used in this hash table was chaining, which added the colliding data to the colliding position's linked list.

The Variables view of the current hashtable object can be seen below:

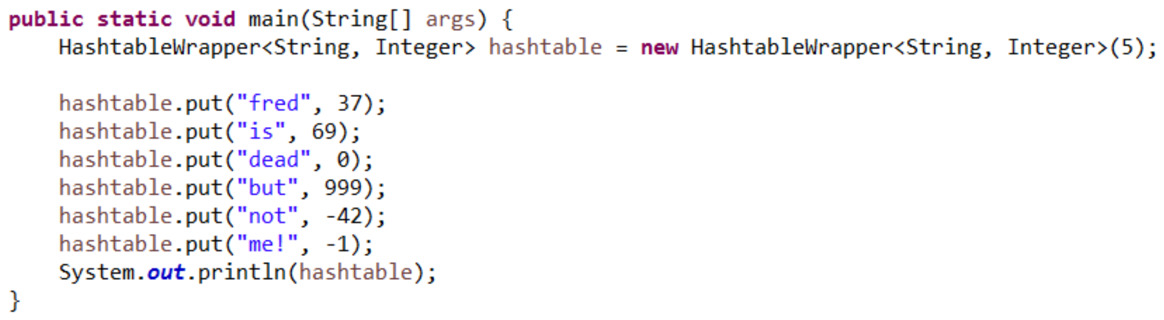


Name	Value
args	String[0] (id=21)
hashtable	HashtableWrapper<S,T> (id=24)
count	5
entrySet	null
keySet	null
loadFactor	0.75
modCount	6
table	Hashtable$Entry<K,V>[11] (id=25)
[0]	Hashtable$Entry<K,V> (id=28)
[1]	null
[2]	null
[3]	null



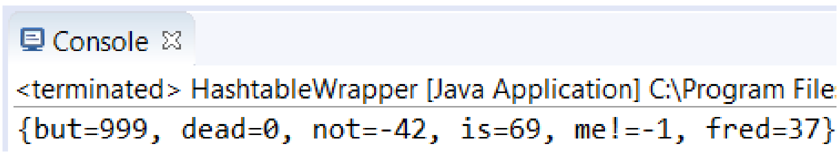
[4]	Hashtable$Entry<K,V> (id=47)
hash	109267
key	"not" (id=50)
next	Hashtable$Entry<K,V> (id=30)
value	Integer (id=51)
[5]	Hashtable$Entry<K,V> (id=31)
[6]	null
[7]	null
[8]	null
[9]	null
[10]	Hashtable$Entry<K,V> (id=32)
threshold	8
values	null

Finally, this screenshot shows final update of the main method, again, using the void ‘put’ method inherited from Hashtable and the final key/value pair of “me”/-1:



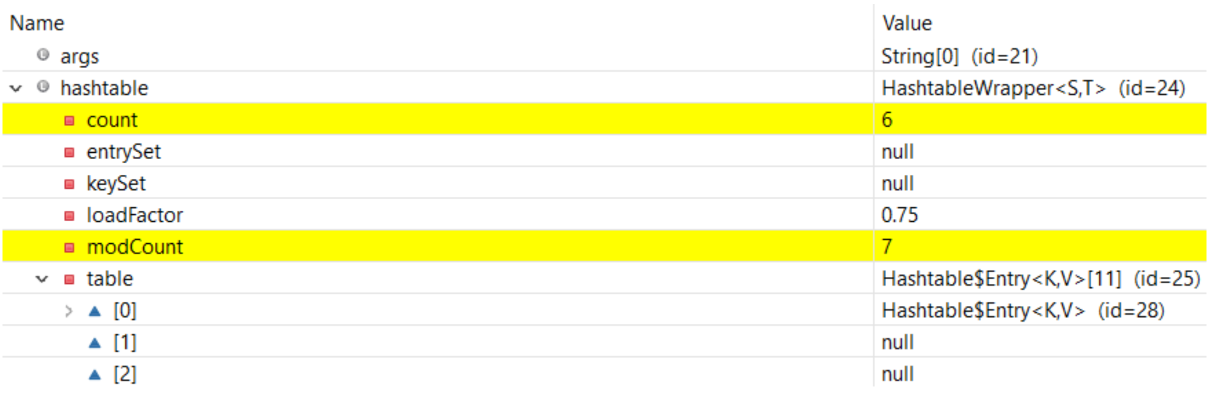
```
public static void main(String[] args) {
    HashtableWrapper<String, Integer> hashtable = new HashtableWrapper<String, Integer>(5);

    hashtable.put("fred", 37);
    hashtable.put("is", 69);
    hashtable.put("dead", 0);
    hashtable.put("but", 999);
    hashtable.put("not", -42);
    hashtable.put("me!", -1);
    System.out.println(hashtable);
}
```

Here is the console output upon running the updated *HashtableWrapper.java:* It can be observed that the hashtable now contains the final key/value pair.

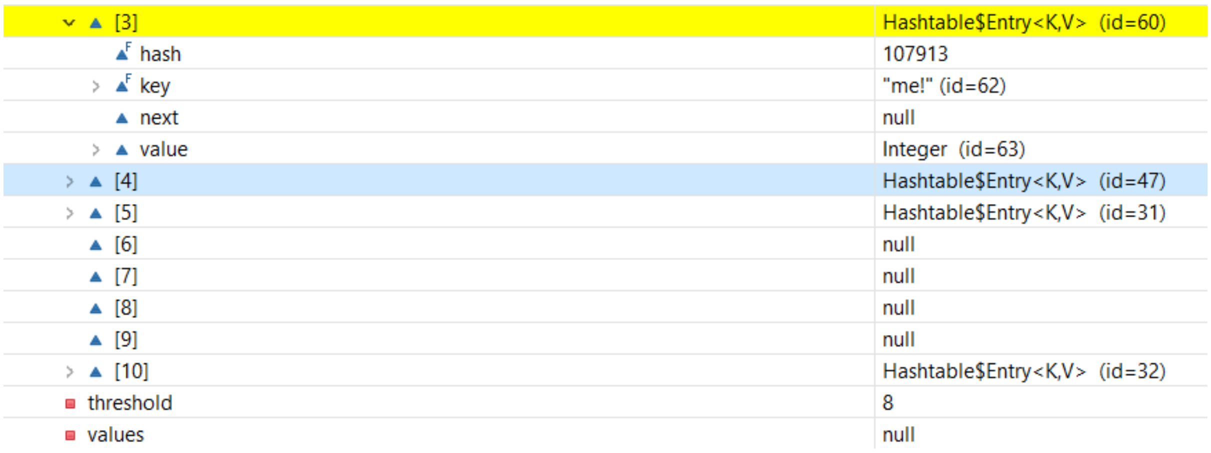


```
Console ☒
<terminated> HashtableWrapper [Java Application] C:\Program File
{but=999, dead=0, not=-42, is=69, me!=-1, fred=37}
```

Upon inspecting the Variables view of the hashtable, it be observed that the count and the modCount of the hashtable have again both increased by one, with the addition of the sixth and final key/value pair into the hashtable: The new key/value pair has been inserted into index[3]. This is because the hash of “me!”/-1 can be seen to be 107913, which has an 11th modulus of 3, resulting in “me”/-1 being placed in index[3].



Name	Value
args	String[0] (id=21)
hashtable	HashtableWrapper<S,T> (id=24)
count	6
entrySet	null
keySet	null
loadFactor	0.75
modCount	7
table	Hashtable$Entry<K,V>[11] (id=25)
[0]	Hashtable$Entry<K,V> (id=28)
[1]	null
[2]	null



[3]	Hashtable$Entry<K,V> (id=60)
hash	107913
key	"me!" (id=62)
next	null
value	Integer (id=63)
[4]	Hashtable$Entry<K,V> (id=47)
[5]	Hashtable$Entry<K,V> (id=31)
[6]	null
[7]	null
[8]	null
[9]	null
[10]	Hashtable$Entry<K,V> (id=32)
threshold	8
values	null

Tutorial Ten

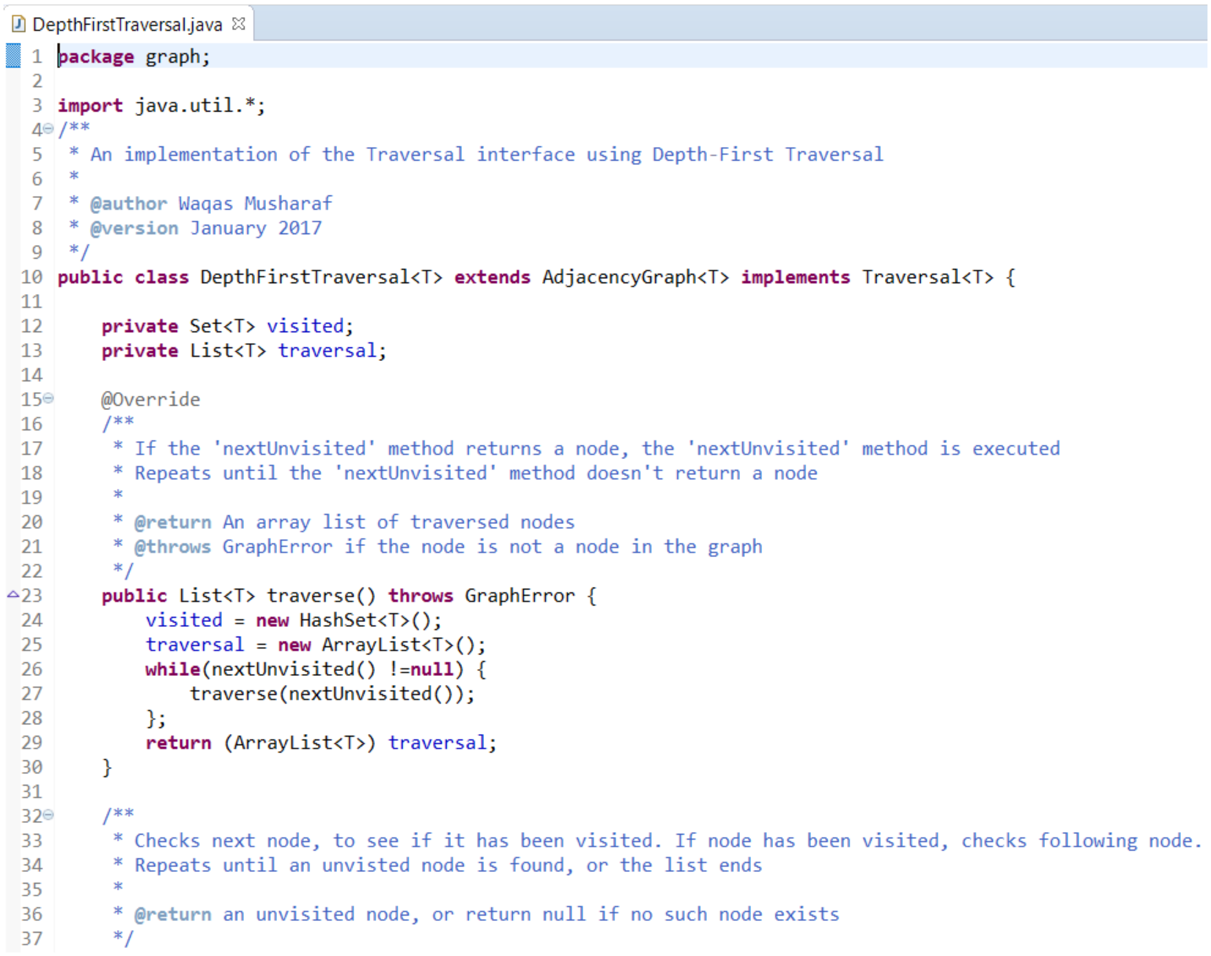
Logbook Exercise:

1) Implement the Traversal interface using depth-first traversal.

Answer:

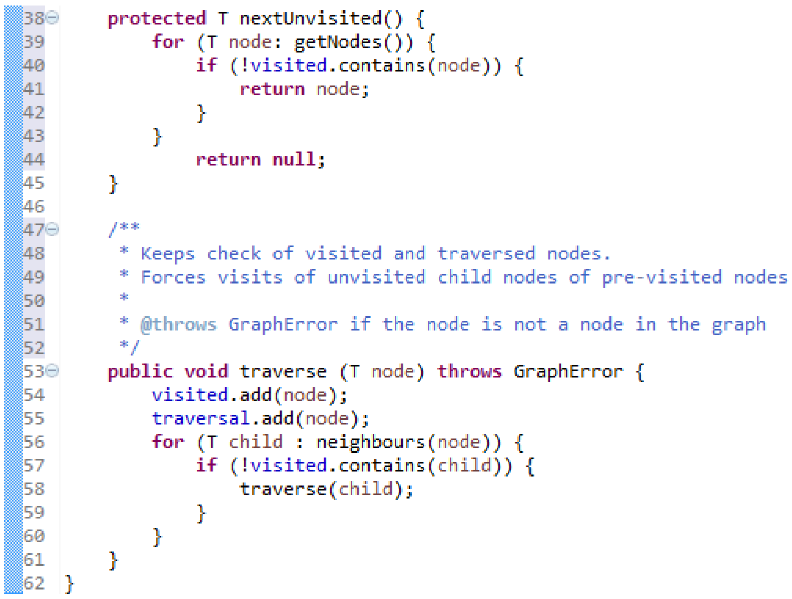
1)

DepthFirstTraversal.java class, implemented using the Traversal Interface:



DepthFirstTraversal.java

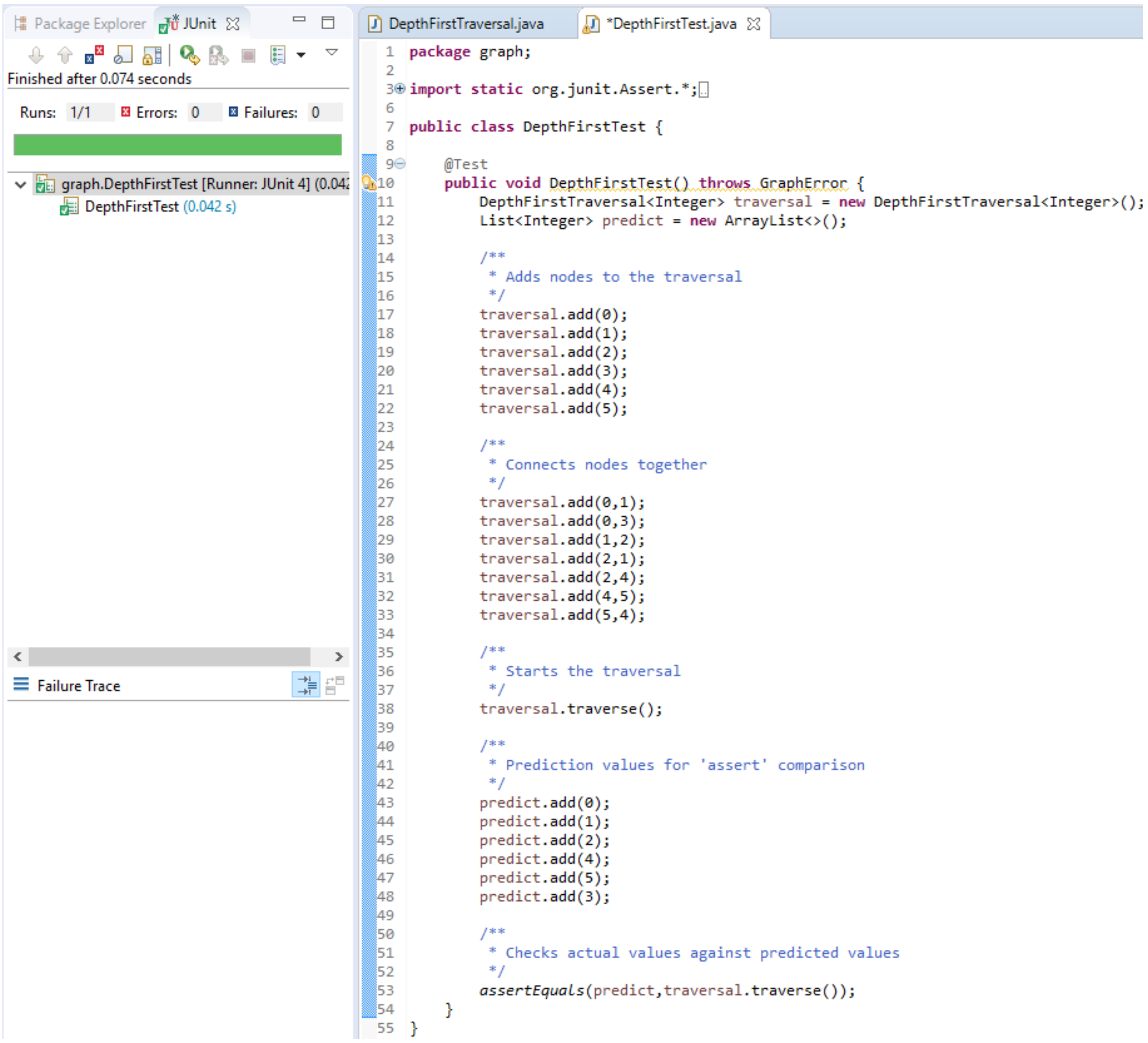
```java
package graph;

import java.util.*;
/**
 * An implementation of the Traversal interface using Depth-First Traversal
 *
 * @author Waqas Musharaf
 * @version January 2017
 */
public class DepthFirstTraversal<T> extends AdjacencyGraph<T> implements Traversal<T> {

    private Set<T> visited;
    private List<T> traversal;

    @Override
    /**
     * If the 'nextUnvisited' method returns a node, the 'nextUnvisited' method is executed
     * Repeats until the 'nextUnvisited' method doesn't return a node
     *
     * @return An array list of traversed nodes
     * @throws GraphError if the node is not a node in the graph
     */
    public List<T> traverse() throws GraphError {
        visited = new HashSet<T>();
        traversal = new ArrayList<T>();
        while(nextUnvisited() !=null) {
            traverse(nextUnvisited());
        };
        return (ArrayList<T>) traversal;
    }

    /**
     * Checks next node, to see if it has been visited. If node has been visited, checks following node.
     * Repeats until an unvisted node is found, or the list ends
     *
     * @return an unvisited node, or return null if no such node exists
     */
```



```java
    protected T nextUnvisited() {
        for (T node: getNodes()) {
            if (!visited.contains(node)) {
                return node;
            }
        }
            return null;
    }

    /**
     * Keeps check of visited and traversed nodes.
     * Forces visits of unvisited child nodes of pre-visited nodes
     *
     * @throws GraphError if the node is not a node in the graph
     */
    public void traverse (T node) throws GraphError {
        visited.add(node);
        traversal.add(node);
        for (T child : neighbours(node)) {
            if (!visited.contains(child)) {
                traverse(child);
            }
        }
    }
}
```

JUnit tests for the DepthFirstTraversal.java class, using DepthFirstTest.java:

The nodes and connections, as well as predicted node values are sourced from the Week 10 lecture notes.



```
Finished after 0.074 seconds
Runs: 1/1    Errors: 0    Failures: 0
graph.DepthFirstTest [Runner: JUnit 4] (0.042
    DepthFirstTest (0.042 s)
```

```java
package graph;

import static org.junit.Assert.*;

public class DepthFirstTest {

    @Test
    public void DepthFirstTest() throws GraphError {
        DepthFirstTraversal<Integer> traversal = new DepthFirstTraversal<Integer>();
        List<Integer> predict = new ArrayList<>();

        /**
         * Adds nodes to the traversal
         */
        traversal.add(0);
        traversal.add(1);
        traversal.add(2);
        traversal.add(3);
        traversal.add(4);
        traversal.add(5);

        /**
         * Connects nodes together
         */
        traversal.add(0,1);
        traversal.add(0,3);
        traversal.add(1,2);
        traversal.add(2,1);
        traversal.add(2,4);
        traversal.add(4,5);
        traversal.add(5,4);

        /**
         * Starts the traversal
         */
        traversal.traverse();

        /**
         * Prediction values for 'assert' comparison
         */
        predict.add(0);
        predict.add(1);
        predict.add(2);
        predict.add(4);
        predict.add(5);
        predict.add(3);

        /**
         * Checks actual values against predicted values
         */
        assertEquals(predict,traversal.traverse());
    }
}
```

From the screenshot above, it can be observed that the JUnit test has run successfully and the predicted values were correct, when compared to the actual values.

Tutorial Eleven

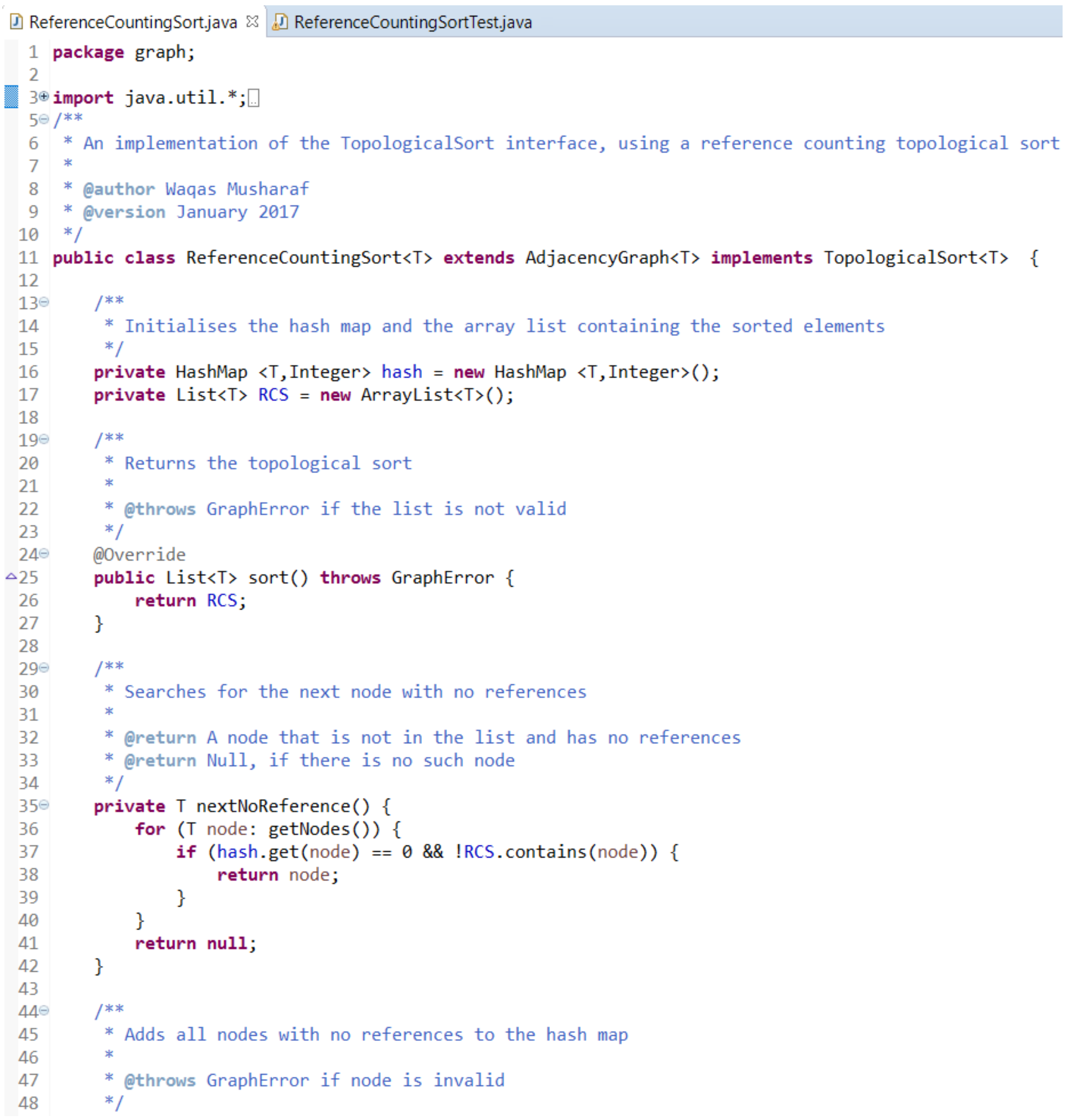
Logbook Exercise:

1) Implement the TopologicalSort interface, using a reference counting topological sort. The getSort() method should return a List<T>, containing a topological sort of the nodes in the graph.

Answer:

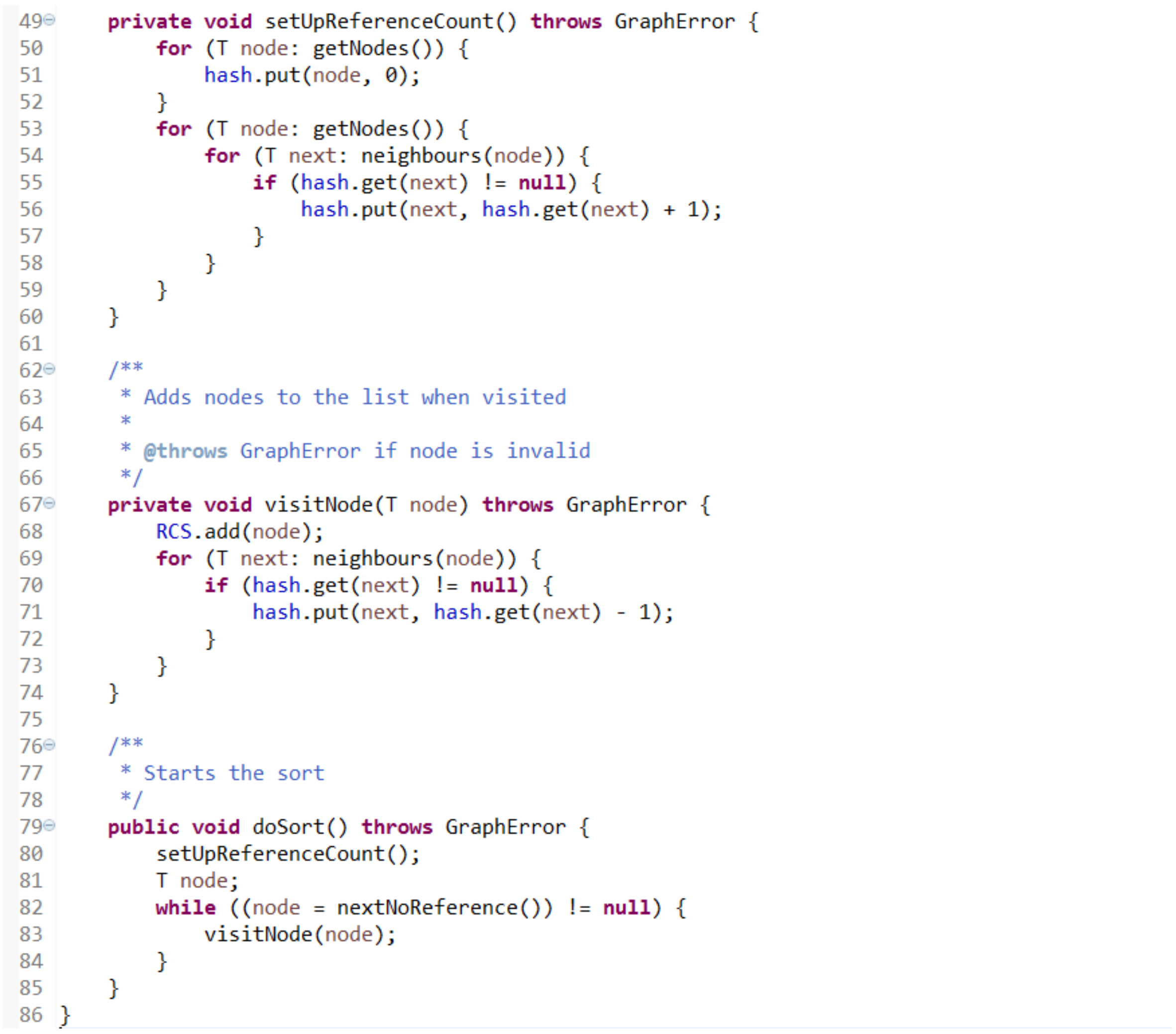
1)

ReferenceCountingSort.java class, implemented using the TopologicalSort Interface:



ReferenceCountingSort.java ☒ ReferenceCountingSortTest.java

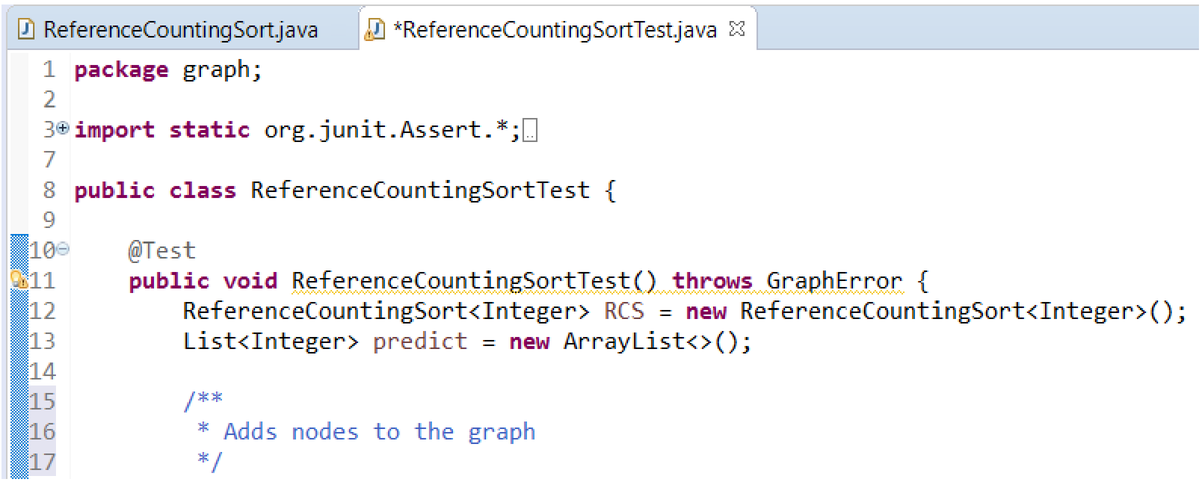
```java
package graph;

import java.util.*;
/**
 * An implementation of the TopologicalSort interface, using a reference counting topological sort
 *
 * @author Waqas Musharaf
 * @version January 2017
 */
public class ReferenceCountingSort<T> extends AdjacencyGraph<T> implements TopologicalSort<T>  {

    /**
     * Initialises the hash map and the array list containing the sorted elements
     */
    private HashMap <T,Integer> hash = new HashMap <T,Integer>();
    private List<T> RCS = new ArrayList<T>();

    /**
     * Returns the topological sort
     *
     * @throws GraphError if the list is not valid
     */
    @Override
    public List<T> sort() throws GraphError {
        return RCS;
    }

    /**
     * Searches for the next node with no references
     *
     * @return A node that is not in the list and has no references
     * @return Null, if there is no such node
     */
    private T nextNoReference() {
        for (T node: getNodes()) {
            if (hash.get(node) == 0 && !RCS.contains(node)) {
                return node;
            }
        }
        return null;
    }

    /**
     * Adds all nodes with no references to the hash map
     *
     * @throws GraphError if node is invalid
     */
```



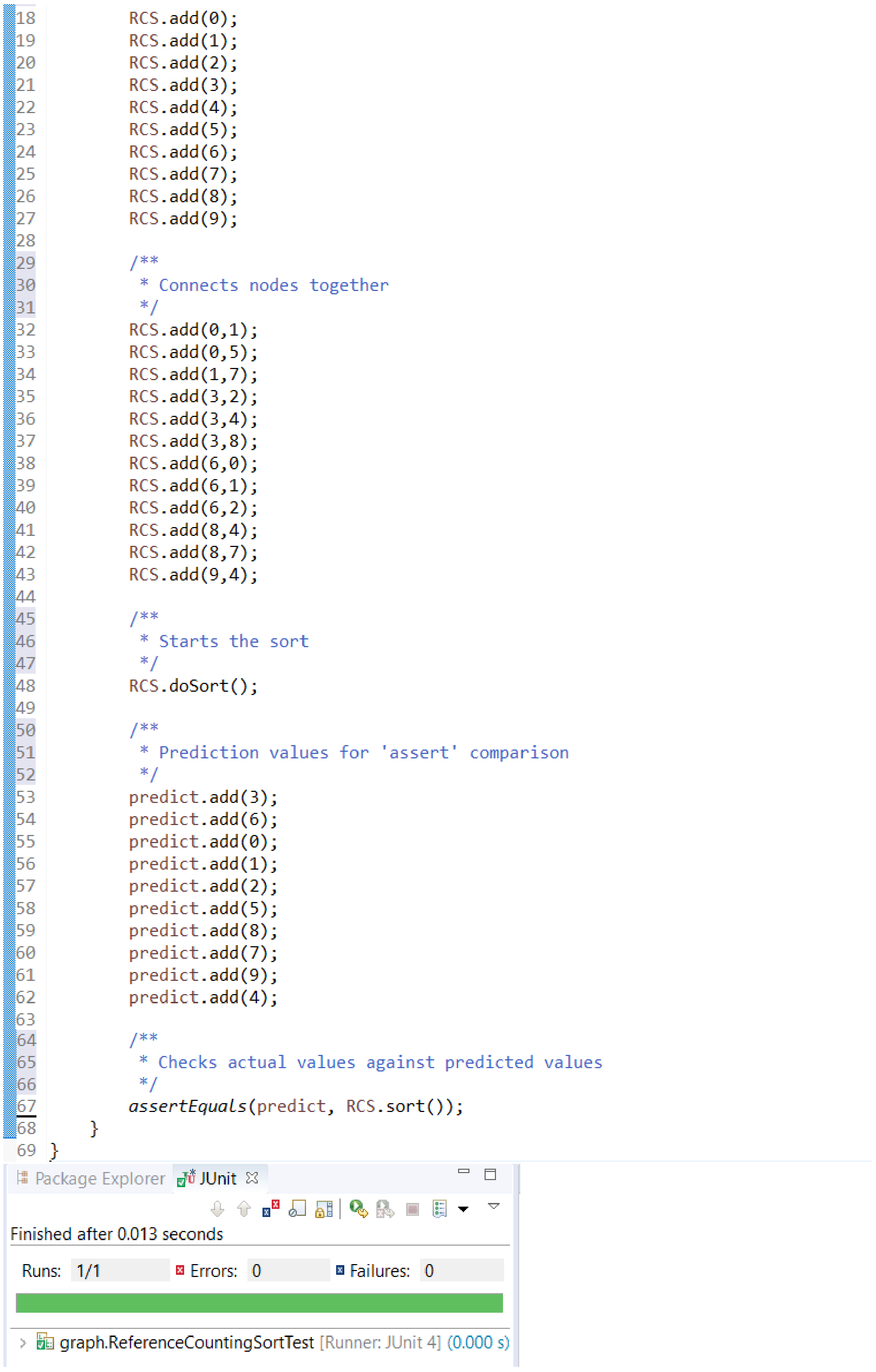
```
    private void setUpReferenceCount() throws GraphError {
        for (T node: getNodes()) {
            hash.put(node, 0);
        }
        for (T node: getNodes()) {
            for (T next: neighbours(node)) {
                if (hash.get(next) != null) {
                    hash.put(next, hash.get(next) + 1);
                }
            }
        }
    }

    /**
     * Adds nodes to the list when visited
     *
     * @throws GraphError if node is invalid
     */
    private void visitNode(T node) throws GraphError {
        RCS.add(node);
        for (T next: neighbours(node)) {
            if (hash.get(next) != null) {
                hash.put(next, hash.get(next) - 1);
            }
        }
    }

    /**
     * Starts the sort
     */
    public void doSort() throws GraphError {
        setUpReferenceCount();
        T node;
        while ((node = nextNoReference()) != null) {
            visitNode(node);
        }
    }
}
```

JUnit tests for the ReferenceCountingSort.java class, using ReferenceCountingSortTest.java:

The nodes and connections, as well as predicted node values are sourced from the Week 11 lecture notes.



```
package graph;

import static org.junit.Assert.*;

public class ReferenceCountingSortTest {

    @Test
    public void ReferenceCountingSortTest() throws GraphError {
        ReferenceCountingSort<Integer> RCS = new ReferenceCountingSort<Integer>();
        List<Integer> predict = new ArrayList<>();

        /**
         * Adds nodes to the graph
         */
```



```
        RCS.add(0);
        RCS.add(1);
        RCS.add(2);
        RCS.add(3);
        RCS.add(4);
        RCS.add(5);
        RCS.add(6);
        RCS.add(7);
        RCS.add(8);
        RCS.add(9);

        /**
         * Connects nodes together
         */
        RCS.add(0,1);
        RCS.add(0,5);
        RCS.add(1,7);
        RCS.add(3,2);
        RCS.add(3,4);
        RCS.add(3,8);
        RCS.add(6,0);
        RCS.add(6,1);
        RCS.add(6,2);
        RCS.add(8,4);
        RCS.add(8,7);
        RCS.add(9,4);

        /**
         * Starts the sort
         */
        RCS.doSort();

        /**
         * Prediction values for 'assert' comparison
         */
        predict.add(3);
        predict.add(6);
        predict.add(0);
        predict.add(1);
        predict.add(2);
        predict.add(5);
        predict.add(8);
        predict.add(7);
        predict.add(9);
        predict.add(4);

        /**
         * Checks actual values against predicted values
         */
        assertEquals(predict, RCS.sort());
    }
}
```

```
Finished after 0.013 seconds

Runs: 1/1    Errors: 0    Failures: 0

graph.ReferenceCountingSortTest [Runner: JUnit 4] (0.000 s)
```

Due to Week 12 being a Consolidation Week, there were no lecture or tutorial sessions and therefore no logbook exercises to complete. There was, however, a self-assessment form completed during Week 12, which can be seen below:

Self-Assessment Two

This self-assessment was completed during Week 12, Consolidation Week. Some work or comments may have changed since the time of writing.

Week(s)	Overall	Documentation	Structure	Naming	Testing	Functionality
9 Hashtables	**A+**	**-**	**-**	**-**	**-**	**-**
10 Depth-First Traversal	**A+**	**A+**	**A+**	**A+**	**A**	**A**
11 Reference-Counting Topological Sort	**A**	**B**	**A**	**B**	**A**	**A**

Assessment Criterion	Grade
Answers to flagged logbook questions	**A+**
Answers to other practical questions	**C**
Other practical work	**C**
Understanding of the module material to date	**A**
Level of self-reflection & evaluation	**A+**
Participation in timetabled activities	**B**
Time spent outside timetabled classes	**A**

Comments:

Week 9: I believe that I have explained the hashtable's behaviour very well, as I covered many sub-topics such as the hashtable internal array, load values, thresholds, hash functions, address collisions and memory management approaches in good detail. I have also provided numerous screenshots of the hashtable object's internal structure throughout the exercise and explained the process of each stage. Overall, I believe this question was answered strongly and in depth.

Week 10: I believe that the quality of documentation for this exercise was very high, as I had many comments throughout my program and test code. Structure and naming is, in my opinion, strong throughout the code. I also conducted JUnit tests of the program and showed screenshots of both the test class and the JUnit tests running. The testing could have been slightly improved by adding more tests, but I didn't feel that this was necessary. The program's functionality was high, as it performed as expected during the tests.

Week 11: I believe that the quality of documentation for this exercise was satisfactory, it could have been improved with additional comments in the extracted code, and I should attempt to add these at a later date. Structure and naming is, in my opinion, strong throughout the code. I also conducted JUnit tests of the program and showed screenshots of both the test class and the JUnit tests running. The testing could have been slightly improved by adding more tests, but I didn't feel that this was necessary. The program's functionality was good, as it performed as expected during the tests.

Overall, I believe that I have answered the flagged logbook questions strongly, although additional work has been lacking. I should strive to add more additional work to the logbook as I progress with the module. So far, my understanding of this module's material is high, as I have been spending more time on this module than previously. I believe my engagement with the module during lessons is also good, because even though I have missed a few lessons, I attempt to go to most timetabled sessions or other sessions that I can attend if I need to. Since the previous self-assessment, I have been spending more time outside of timetabled lessons to work on this module, and have noticed improvements in my understanding. A target I could set for myself could be to try harder to attend all timetabled sessions and keep working on the module outside of timetabled lessons.

Tutorial Thirteen

Logbook Exercises:

1) Add a new test to the CountTest class that will create two Counters; one that tries to count from 0 to 10, incrementing by one each time, and one that tries to count from 10 to 0, decrementing by one each time. Make sure that both Counters trace their behaviour.

Run this test several times, and answer the following questions. Make sure that you explain your answers:

2) Will the test always terminate? i.e. is it certain that no matter how often you were to run the test it would always end in a finite length of time?

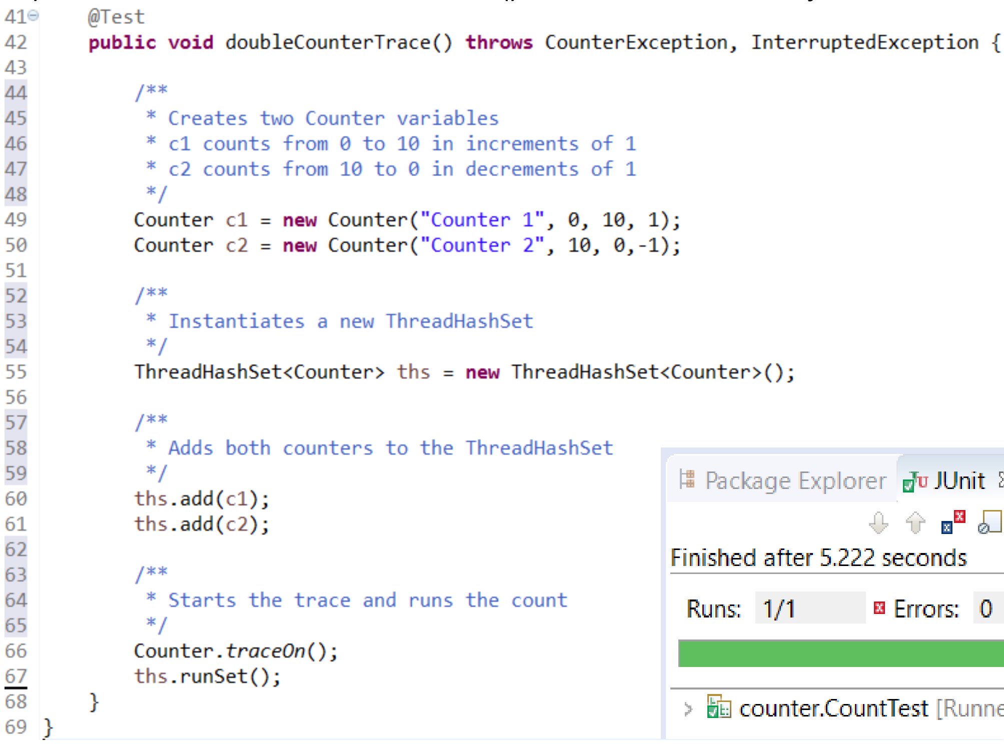
3) What is the shortest possible output for the test, in terms of the number of lines output?

4) What is the largest possible value that the count can reach when the test is run?

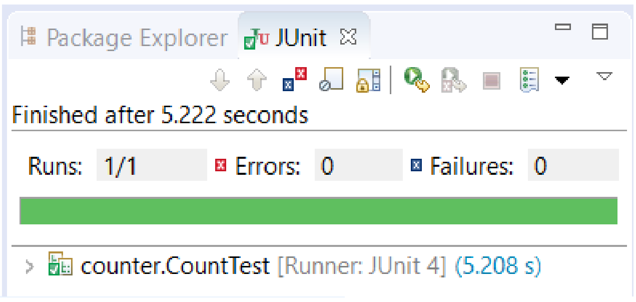
5) What is the lowest possible value that the count can reach when the test is run?

Answers:

1a) New test method doubleCounterTrace() inserted into CountTest.java:



```
    @Test
    public void doubleCounterTrace() throws CounterException, InterruptedException {

        /**
         * Creates two Counter variables
         * c1 counts from 0 to 10 in increments of 1
         * c2 counts from 10 to 0 in decrements of 1
         */
        Counter c1 = new Counter("Counter 1", 0, 10, 1);
        Counter c2 = new Counter("Counter 2", 10, 0,-1);

        /**
         * Instantiates a new ThreadHashSet
         */
        ThreadHashSet<Counter> ths = new ThreadHashSet<Counter>();

        /**
         * Adds both counters to the ThreadHashSet
         */
        ths.add(c1);
        ths.add(c2);

        /**
         * Starts the trace and runs the count
         */
        Counter.traceOn();
        ths.runSet();
    }
}
```

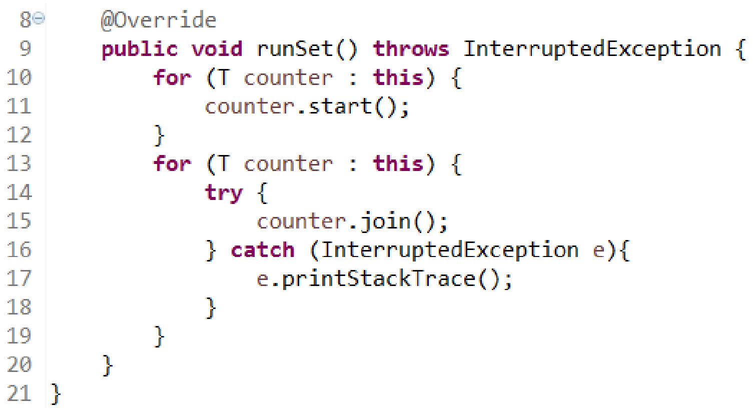


Finished after 5.222 seconds

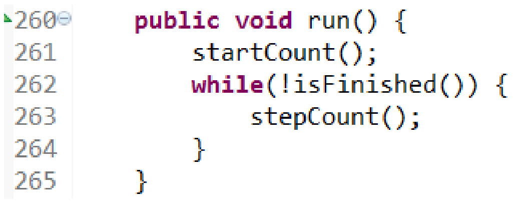
Runs: 1/1 Errors: 0 Failures: 0

counter.CountTest [Runner: JUnit 4] (5.208 s)

1b) runSet() implemented into ThreadHashSet.java and run() method implemented into Counter.java:



```java
    @Override
    public void runSet() throws InterruptedException {
        for (T counter : this) {
            counter.start();
        }
        for (T counter : this) {
            try {
                counter.join();
            } catch (InterruptedException e){
                e.printStackTrace();
            }
        }
    }
}
```



```java
    public void run() {
        startCount();
        while(!isFinished()) {
            stepCount();
        }
    }
```

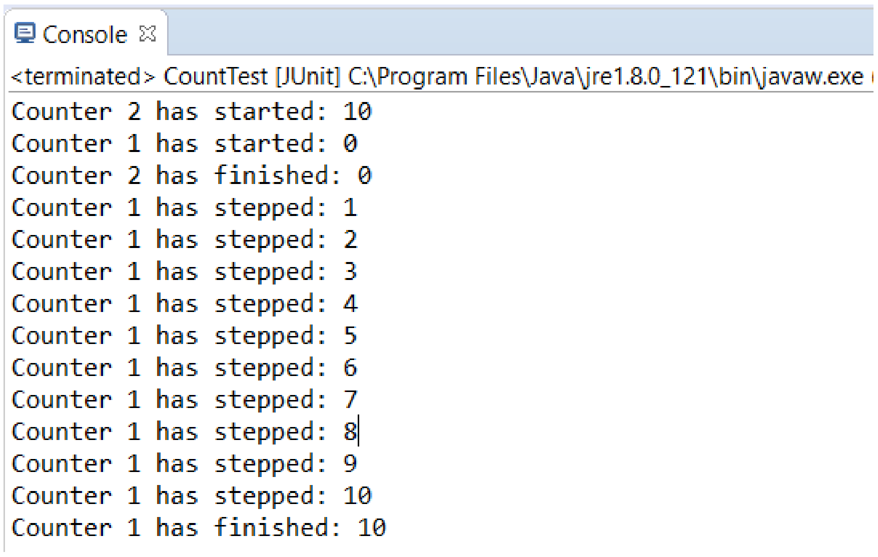
2) Will the test always terminate? i.e. is it certain that no matter how often you were to run the test it would always end in a finite length of time?

Yes, the test will always terminate. Both counters start at the other counter's end-point, resulting in one counter terminating very quickly as soon as the other counter begins to count. For example, if counter 1 has a start point of 0 and an end-point of 10 and counter 2 has the opposite, and one counter begins counting, as soon as the counting counter switches, one of the counters will instantly be at their end-point, allowing that counter to terminate. Once a counter has terminated, there will be no more resource sharing required, and the non-terminated counter can increment/decrement until it too hits it end-point, upon which it will also terminate.

3) What is the shortest possible output for the test, in terms of the number of lines output?

The shortest possible output for the test is 14 lines. This is because each counter takes one line each start, and also takes one line each to end. If one counter terminates immediately due to the other counter beginning to step and therefore arriving at the first counter's end-point immediately, the first counter will take zero lines to step. That leaves the final counter to perform its 10 steps to get from its start-point to its end-point. This results in a total of 2+2+10 lines, which equals a shortest possible output of 14 lines.

Here is an example of the shortest possible output, from one of the tests I performed:



```
<terminated> CountTest [JUnit] C:\Program Files\Java\jre1.8.0_121\bin\javaw.exe
Counter 2 has started: 10
Counter 1 has started: 0
Counter 2 has finished: 0
Counter 1 has stepped: 1
Counter 1 has stepped: 2
Counter 1 has stepped: 3
Counter 1 has stepped: 4
Counter 1 has stepped: 5
Counter 1 has stepped: 6
Counter 1 has stepped: 7
Counter 1 has stepped: 8
Counter 1 has stepped: 9
Counter 1 has stepped: 10
Counter 1 has finished: 10
```

4) What is the largest possible value that the count can reach when the test is run?

The largest possible value is theoretically 11. This could occur when the following occurs:

- Counter 1 starts first with value 0
- Counter 2 starts next with value 10
- Counter 1 automatically increases to value 10
- Counter 1 initiates increment of 1, assuming the value is still 0
- Counter 1 increments to 11

Unfortunately, after many tests I couldn't get this scenario to occur, which is why I say 11 is the *theoretical* maximum value.

5) What is the lowest possible value that the count can reach when the test is run?

The lowest possible value is theoretically -1. This could occur when the following occurs:

- Counter 2 starts first with value 10
- Counter 1 starts next with value 0
- Counter 2 automatically decreases to value 0
- Counter 2 initiates decrement of 1, assuming the value is still 10
- Counter 2 decrements to -1

Unfortunately, after many tests I couldn't get this scenario to occur, which is why I say -1 is the *theoretical* minimum value.

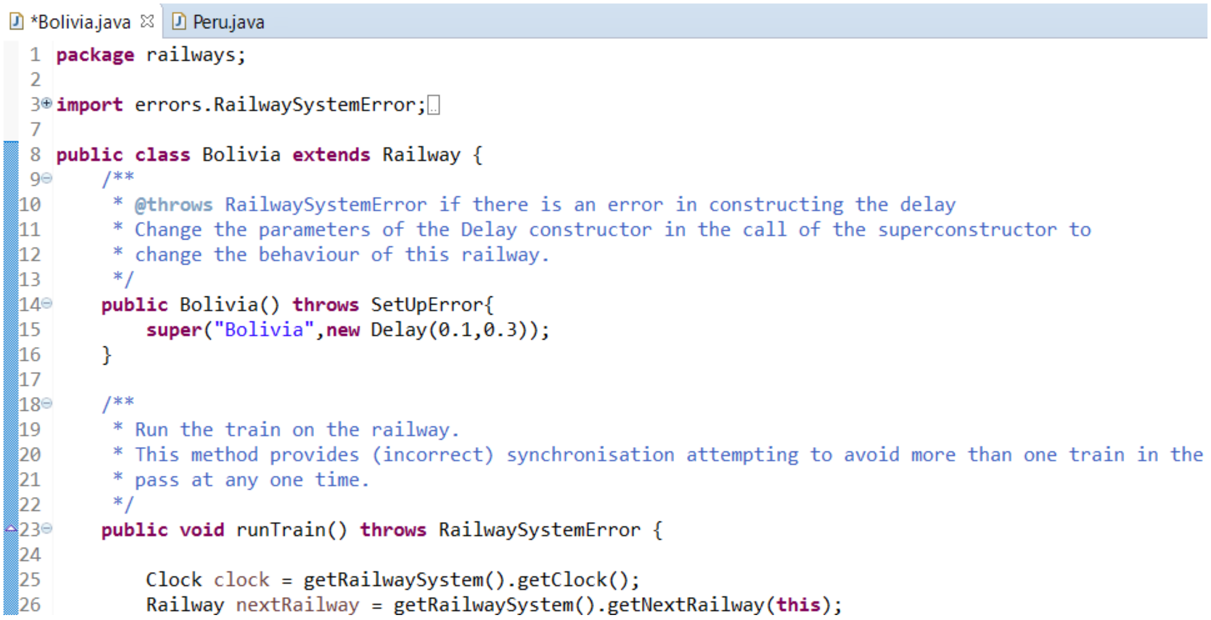
Tutorial Fourteen

Logbook Exercise:

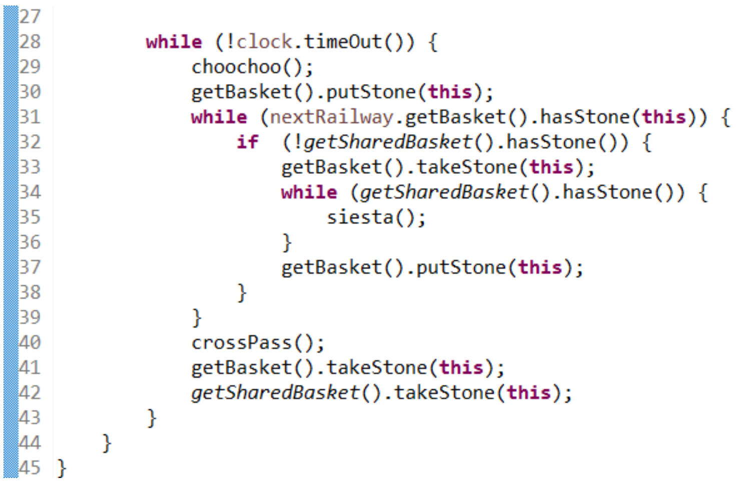
1) The train companies sacked the consultant, and hired a recent Huddersfield graduate. She immediately saw the solution, having paid careful attention to Hugh's lecture on Dekker's algorithm. Implement her solution

Answer:

1a) Implementation of Dekker's Algorithm within Bolivia.java:

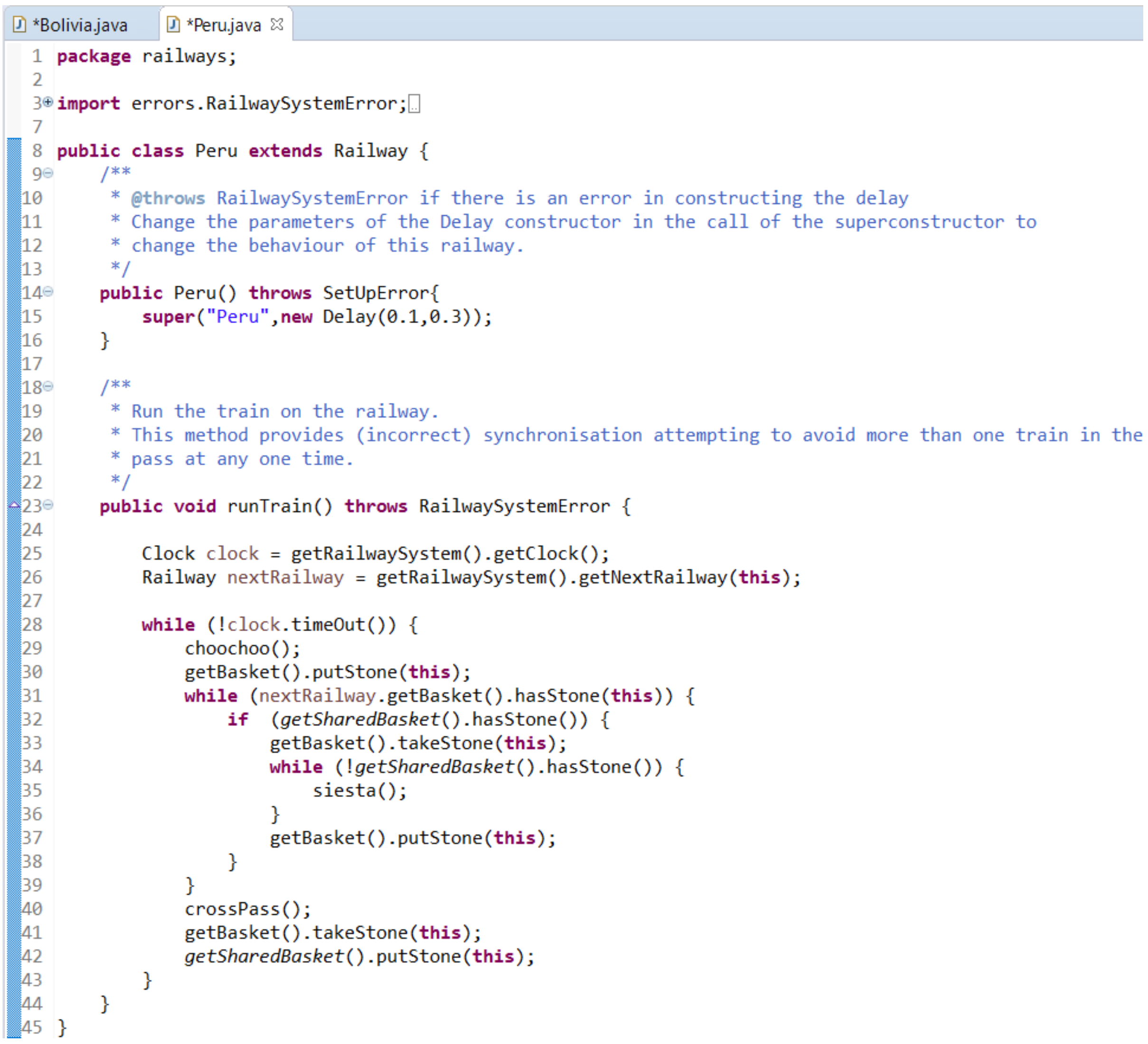


```
package railways;

import errors.RailwaySystemError;

public class Bolivia extends Railway {
    /**
     * @throws RailwaySystemError if there is an error in constructing the delay
     * Change the parameters of the Delay constructor in the call of the superconstructor to
     * change the behaviour of this railway.
     */
    public Bolivia() throws SetUpError{
        super("Bolivia",new Delay(0.1,0.3));
    }

    /**
     * Run the train on the railway.
     * This method provides (incorrect) synchronisation attempting to avoid more than one train in the
     * pass at any one time.
     */
    public void runTrain() throws RailwaySystemError {

        Clock clock = getRailwaySystem().getClock();
        Railway nextRailway = getRailwaySystem().getNextRailway(this);
```



```java

        while (!clock.timeOut()) {
            choochoo();
            getBasket().putStone(this);
            while (nextRailway.getBasket().hasStone(this)) {
                if  (!getSharedBasket().hasStone()) {
                    getBasket().takeStone(this);
                    while (getSharedBasket().hasStone()) {
                        siesta();
                    }
                    getBasket().putStone(this);
                }
            }
            crossPass();
            getBasket().takeStone(this);
            getSharedBasket().takeStone(this);
        }
    }
}
```

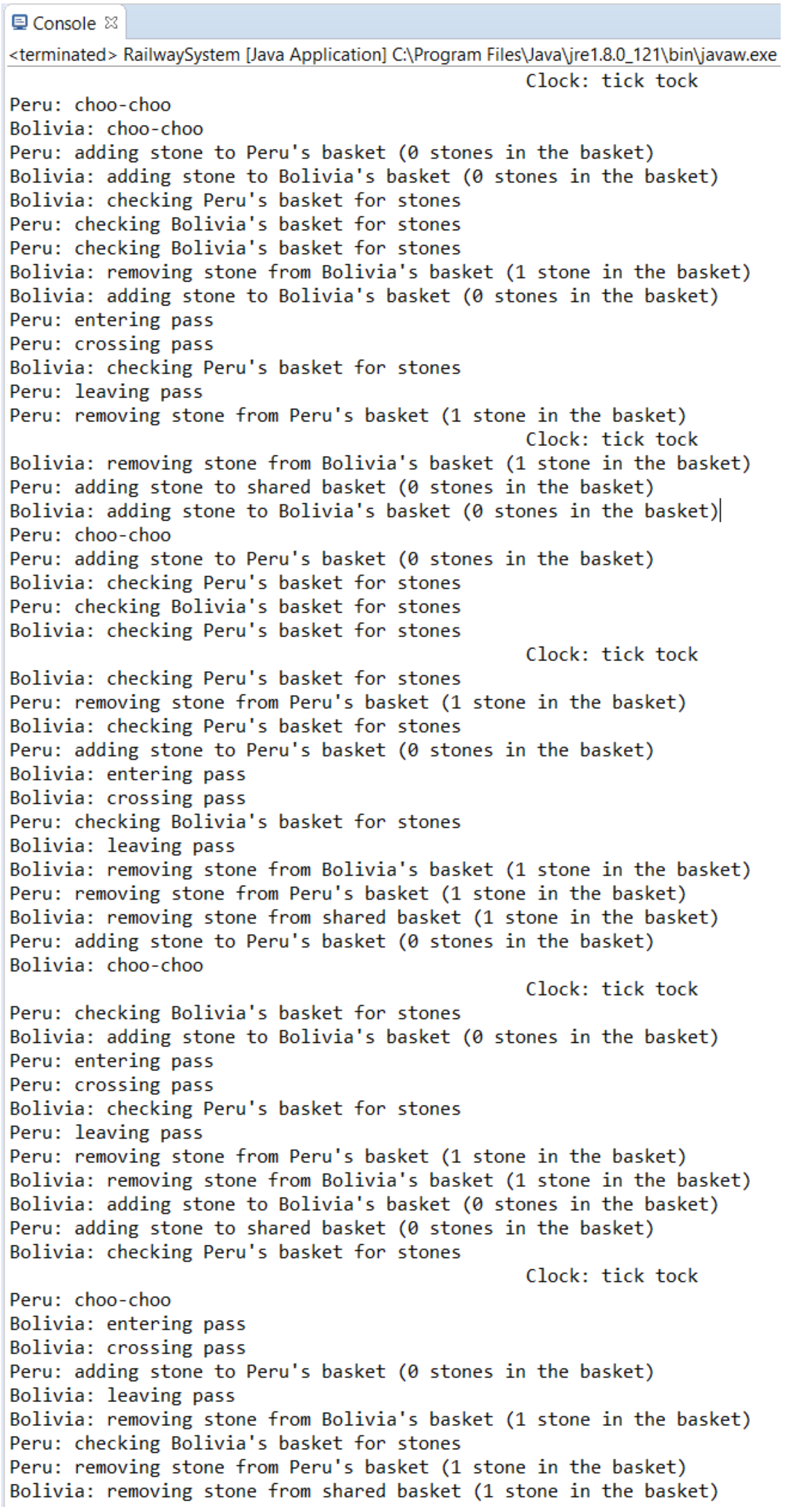
1b) Implementation of Dekker's Algorithm within Peru.java:



*Bolivia.java   *Peru.java

```java
package railways;

import errors.RailwaySystemError;

public class Peru extends Railway {
    /**
     * @throws RailwaySystemError if there is an error in constructing the delay
     * Change the parameters of the Delay constructor in the call of the superconstructor to
     * change the behaviour of this railway.
     */
    public Peru() throws SetUpError{
        super("Peru",new Delay(0.1,0.3));
    }

    /**
     * Run the train on the railway.
     * This method provides (incorrect) synchronisation attempting to avoid more than one train in the
     * pass at any one time.
     */
    public void runTrain() throws RailwaySystemError {

        Clock clock = getRailwaySystem().getClock();
        Railway nextRailway = getRailwaySystem().getNextRailway(this);

        while (!clock.timeOut()) {
            choochoo();
            getBasket().putStone(this);
            while (nextRailway.getBasket().hasStone(this)) {
                if  (getSharedBasket().hasStone()) {
                    getBasket().takeStone(this);
                    while (!getSharedBasket().hasStone()) {
                        siesta();
                    }
                    getBasket().putStone(this);
                }
            }
            crossPass();
            getBasket().takeStone(this);
            getSharedBasket().putStone(this);
        }
    }
}
```

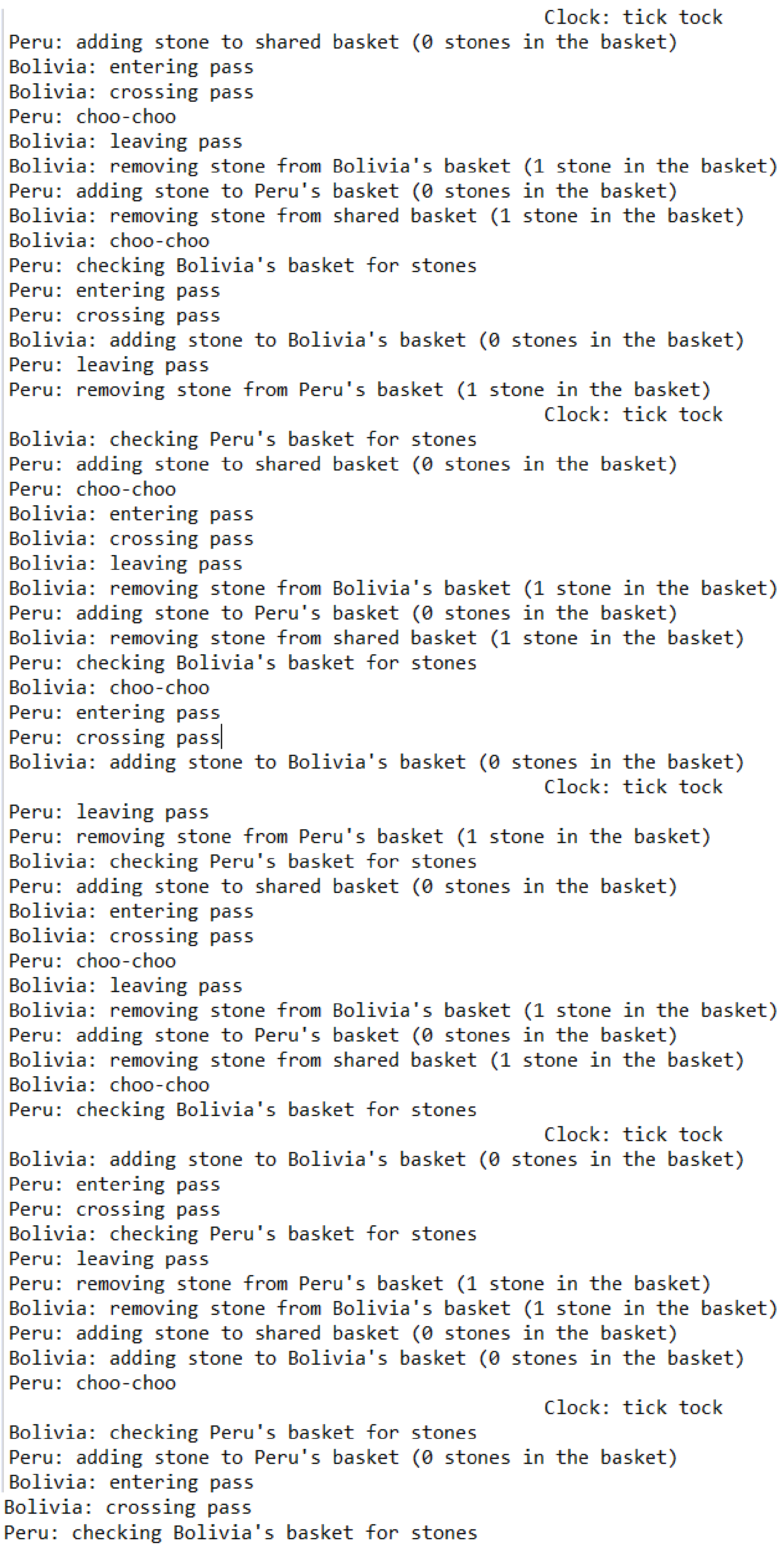
1c) Full console output upon running RailwaySystem.java after updating Bolivia.java and Peru.java (RailwaySystem.java was provided complete):



Console

```
<terminated> RailwaySystem [Java Application] C:\Program Files\Java\jre1.8.0_121\bin\javaw.exe
                                                Clock: tick tock
Peru: choo-choo
Bolivia: choo-choo
Peru: adding stone to Peru's basket (0 stones in the basket)
Bolivia: adding stone to Bolivia's basket (0 stones in the basket)
Bolivia: checking Peru's basket for stones
Peru: checking Bolivia's basket for stones
Peru: checking Bolivia's basket for stones
Bolivia: removing stone from Bolivia's basket (1 stone in the basket)
Bolivia: adding stone to Bolivia's basket (0 stones in the basket)
Peru: entering pass
Peru: crossing pass
Bolivia: checking Peru's basket for stones
Peru: leaving pass
Peru: removing stone from Peru's basket (1 stone in the basket)
                                                Clock: tick tock
Bolivia: removing stone from Bolivia's basket (1 stone in the basket)
Peru: adding stone to shared basket (0 stones in the basket)
Bolivia: adding stone to Bolivia's basket (0 stones in the basket)
Peru: choo-choo
Peru: adding stone to Peru's basket (0 stones in the basket)
Bolivia: checking Peru's basket for stones
Peru: checking Bolivia's basket for stones
Bolivia: checking Peru's basket for stones
                                                Clock: tick tock
Bolivia: checking Peru's basket for stones
Peru: removing stone from Peru's basket (1 stone in the basket)
Bolivia: checking Peru's basket for stones
Peru: adding stone to Peru's basket (0 stones in the basket)
Bolivia: entering pass
Bolivia: crossing pass
Peru: checking Bolivia's basket for stones
Bolivia: leaving pass
Bolivia: removing stone from Bolivia's basket (1 stone in the basket)
Peru: removing stone from Peru's basket (1 stone in the basket)
Bolivia: removing stone from shared basket (1 stone in the basket)
Peru: adding stone to Peru's basket (0 stones in the basket)
Bolivia: choo-choo
                                                Clock: tick tock
Peru: checking Bolivia's basket for stones
Bolivia: adding stone to Bolivia's basket (0 stones in the basket)
Peru: entering pass
Peru: crossing pass
Bolivia: checking Peru's basket for stones
Peru: leaving pass
Peru: removing stone from Peru's basket (1 stone in the basket)
Bolivia: removing stone from Bolivia's basket (1 stone in the basket)
Bolivia: adding stone to Bolivia's basket (0 stones in the basket)
Peru: adding stone to shared basket (0 stones in the basket)
Bolivia: checking Peru's basket for stones
                                                Clock: tick tock
Peru: choo-choo
Bolivia: entering pass
Bolivia: crossing pass
Peru: adding stone to Peru's basket (0 stones in the basket)
Bolivia: leaving pass
Bolivia: removing stone from Bolivia's basket (1 stone in the basket)
Peru: checking Bolivia's basket for stones
Peru: removing stone from Peru's basket (1 stone in the basket)
Bolivia: removing stone from shared basket (1 stone in the basket)
```

```
Peru: adding stone to Peru's basket (0 stones in the basket)
Bolivia: choo-choo
                                                Clock: tick tock
Peru: checking Bolivia's basket for stones
Bolivia: adding stone to Bolivia's basket (0 stones in the basket)
Peru: entering pass
Peru: crossing pass
Bolivia: checking Peru's basket for stones
Peru: leaving pass
Peru: removing stone from Peru's basket (1 stone in the basket)
Bolivia: removing stone from Bolivia's basket (1 stone in the basket)
Bolivia: adding stone to Bolivia's basket (0 stones in the basket)
Peru: adding stone to shared basket (0 stones in the basket)
Bolivia: checking Peru's basket for stones
Bolivia: entering pass
Bolivia: crossing pass
Peru: choo-choo
Bolivia: leaving pass
Bolivia: removing stone from Bolivia's basket (1 stone in the basket)
                                                Clock: tick tock
Peru: adding stone to Peru's basket (0 stones in the basket)
Bolivia: removing stone from shared basket (1 stone in the basket)
Peru: checking Bolivia's basket for stones
Bolivia: choo-choo
Peru: entering pass
Peru: crossing pass
Bolivia: adding stone to Bolivia's basket (0 stones in the basket)
Peru: leaving pass
Peru: removing stone from Peru's basket (1 stone in the basket)
Bolivia: checking Peru's basket for stones
Peru: adding stone to shared basket (0 stones in the basket)
Peru: choo-choo
Bolivia: entering pass
Bolivia: crossing pass
Peru: adding stone to Peru's basket (0 stones in the basket)
                                                Clock: tick tock
Bolivia: leaving pass
Bolivia: removing stone from Bolivia's basket (1 stone in the basket)
Peru: checking Bolivia's basket for stones
Bolivia: removing stone from shared basket (1 stone in the basket)
Peru: entering pass
Peru: crossing pass
Peru: leaving pass
Peru: removing stone from Peru's basket (1 stone in the basket)
Bolivia: choo-choo
Bolivia: adding stone to Bolivia's basket (0 stones in the basket)
Peru: adding stone to shared basket (0 stones in the basket)
Bolivia: checking Peru's basket for stones
Peru: choo-choo
Bolivia: entering pass
Bolivia: crossing pass
                                                Clock: tick tock
Bolivia: leaving pass
Bolivia: removing stone from Bolivia's basket (1 stone in the basket)
Peru: adding stone to Peru's basket (0 stones in the basket)
Bolivia: removing stone from shared basket (1 stone in the basket)
Peru: checking Bolivia's basket for stones
Bolivia: choo-choo
Peru: entering pass
Peru: crossing pass
Bolivia: adding stone to Bolivia's basket (0 stones in the basket)
Peru: leaving pass
Peru: removing stone from Peru's basket (1 stone in the basket)
Bolivia: checking Peru's basket for stones
```



```
                                                Clock: tick tock
Peru: adding stone to shared basket (0 stones in the basket)
Bolivia: entering pass
Bolivia: crossing pass
Peru: choo-choo
Bolivia: leaving pass
Bolivia: removing stone from Bolivia's basket (1 stone in the basket)
Peru: adding stone to Peru's basket (0 stones in the basket)
Bolivia: removing stone from shared basket (1 stone in the basket)
Bolivia: choo-choo
Peru: checking Bolivia's basket for stones
Peru: entering pass
Peru: crossing pass
Bolivia: adding stone to Bolivia's basket (0 stones in the basket)
Peru: leaving pass
Peru: removing stone from Peru's basket (1 stone in the basket)
                                                Clock: tick tock
Bolivia: checking Peru's basket for stones
Peru: adding stone to shared basket (0 stones in the basket)
Peru: choo-choo
Bolivia: entering pass
Bolivia: crossing pass
Bolivia: leaving pass
Bolivia: removing stone from Bolivia's basket (1 stone in the basket)
Peru: adding stone to Peru's basket (0 stones in the basket)
Bolivia: removing stone from shared basket (1 stone in the basket)
Peru: checking Bolivia's basket for stones
Bolivia: choo-choo
Peru: entering pass
Peru: crossing pass
Bolivia: adding stone to Bolivia's basket (0 stones in the basket)
                                                Clock: tick tock
Peru: leaving pass
Peru: removing stone from Peru's basket (1 stone in the basket)
Bolivia: checking Peru's basket for stones
Peru: adding stone to shared basket (0 stones in the basket)
Bolivia: entering pass
Bolivia: crossing pass
Peru: choo-choo
Bolivia: leaving pass
Bolivia: removing stone from Bolivia's basket (1 stone in the basket)
Peru: adding stone to Peru's basket (0 stones in the basket)
Bolivia: removing stone from shared basket (1 stone in the basket)
Bolivia: choo-choo
Peru: checking Bolivia's basket for stones
                                                Clock: tick tock
Bolivia: adding stone to Bolivia's basket (0 stones in the basket)
Peru: entering pass
Peru: crossing pass
Bolivia: checking Peru's basket for stones
Peru: leaving pass
Peru: removing stone from Peru's basket (1 stone in the basket)
Bolivia: removing stone from Bolivia's basket (1 stone in the basket)
Peru: adding stone to shared basket (0 stones in the basket)
Bolivia: adding stone to Bolivia's basket (0 stones in the basket)
Peru: choo-choo
                                                Clock: tick tock
Bolivia: checking Peru's basket for stones
Peru: adding stone to Peru's basket (0 stones in the basket)
Bolivia: entering pass
Bolivia: crossing pass
Peru: checking Bolivia's basket for stones
```

```
Bolivia: leaving pass
Bolivia: removing stone from Bolivia's basket (1 stone in the basket)
Peru: removing stone from Peru's basket (1 stone in the basket)
Bolivia: removing stone from shared basket (1 stone in the basket)
Peru: adding stone to Peru's basket (0 stones in the basket)
Bolivia: choo-choo
Bolivia: adding stone to Bolivia's basket (0 stones in the basket)
                                                Clock: tick tock
Peru: checking Bolivia's basket for stones
Peru: entering pass
Peru: crossing pass
Bolivia: checking Peru's basket for stones
Peru: leaving pass
Peru: removing stone from Peru's basket (1 stone in the basket)
Bolivia: removing stone from Bolivia's basket (1 stone in the basket)
Peru: adding stone to shared basket (0 stones in the basket)
Bolivia: adding stone to Bolivia's basket (0 stones in the basket)
Peru: choo-choo
Peru: adding stone to Peru's basket (0 stones in the basket)
Bolivia: checking Peru's basket for stones
Bolivia: entering pass
Bolivia: crossing pass
                                                Clock: tick tock
Peru: checking Bolivia's basket for stones
Peru: removing stone from Peru's basket (1 stone in the basket)
Bolivia: leaving pass
Bolivia: removing stone from Bolivia's basket (1 stone in the basket)
Peru: adding stone to Peru's basket (0 stones in the basket)
Bolivia: removing stone from shared basket (1 stone in the basket)
Peru: checking Bolivia's basket for stones
Bolivia: choo-choo
Peru: entering pass
Peru: crossing pass
Bolivia: adding stone to Bolivia's basket (0 stones in the basket)
Peru: leaving pass
Peru: removing stone from Peru's basket (1 stone in the basket)
                                                Clock: tick tock
Bolivia: checking Peru's basket for stones
Peru: adding stone to shared basket (0 stones in the basket)
Bolivia: entering pass
Bolivia: crossing pass
Peru: choo-choo
Bolivia: leaving pass
Bolivia: removing stone from Bolivia's basket (1 stone in the basket)
Bolivia: removing stone from shared basket (1 stone in the basket)
Peru: adding stone to Peru's basket (0 stones in the basket)
Bolivia: choo-choo
Peru: checking Bolivia's basket for stones
                                                Clock: tick tock
Peru: entering pass
Peru: crossing pass
Bolivia: adding stone to Bolivia's basket (0 stones in the basket)
Peru: leaving pass
Peru: removing stone from Peru's basket (1 stone in the basket)
Bolivia: checking Peru's basket for stones
Bolivia: removing stone from Bolivia's basket (1 stone in the basket)
Peru: adding stone to shared basket (0 stones in the basket)
Bolivia: adding stone to Bolivia's basket (0 stones in the basket)
Peru: choo-choo
Peru: adding stone to Peru's basket (0 stones in the basket)
```

```
Bolivia: checking Peru's basket for stones
Peru: checking Bolivia's basket for stones
Bolivia: checking Peru's basket for stones
                                                Clock: tick tock
Peru: removing stone from Peru's basket (1 stone in the basket)
Bolivia: checking Peru's basket for stones
Peru: adding stone to Peru's basket (0 stones in the basket)
Bolivia: entering pass
Bolivia: crossing pass
Bolivia: leaving pass
Bolivia: removing stone from Bolivia's basket (1 stone in the basket)
Peru: checking Bolivia's basket for stones
Bolivia: removing stone from shared basket (1 stone in the basket)
Peru: entering pass
Peru: crossing pass
Peru: leaving pass
Peru: removing stone from Peru's basket (1 stone in the basket)
Bolivia: choo-choo
                                                Clock: tick tock
Peru: adding stone to shared basket (0 stones in the basket)
Bolivia: adding stone to Bolivia's basket (0 stones in the basket)
Peru: choo-choo
Bolivia: checking Peru's basket for stones
Bolivia: entering pass
Bolivia: crossing pass
Peru: adding stone to Peru's basket (0 stones in the basket)
Peru: checking Bolivia's basket for stones
Bolivia: leaving pass
Bolivia: removing stone from Bolivia's basket (1 stone in the basket)
Peru: removing stone from Peru's basket (1 stone in the basket)
                                                Clock: brrrrring!
Peru: adding stone to Peru's basket (0 stones in the basket)
Bolivia: removing stone from shared basket (1 stone in the basket)
Peru: checking Bolivia's basket for stones
Bolivia shut down because time limit was reached.
Peru: entering pass
Peru: crossing pass
Peru: leaving pass
Peru: removing stone from Peru's basket (1 stone in the basket)
Peru: adding stone to shared basket (0 stones in the basket)
Peru shut down because time limit was reached.
```

Tutorial Fifteen

Logbook Exercises:

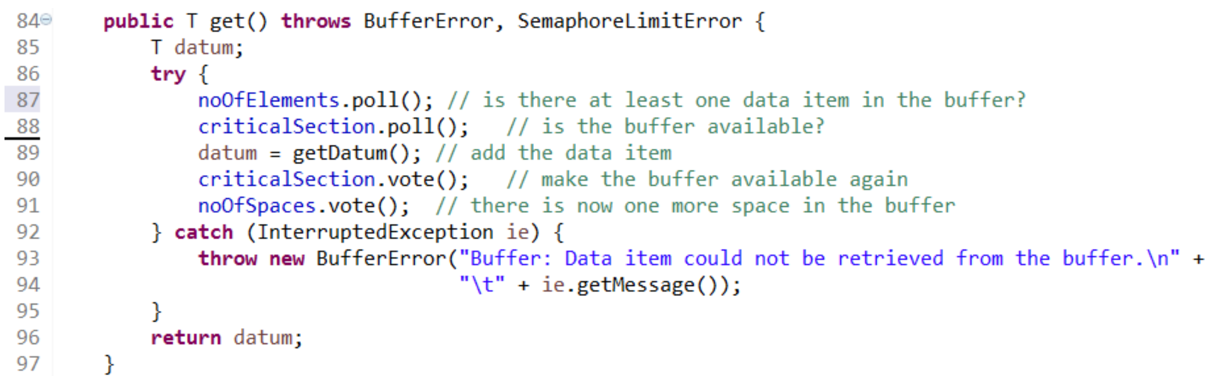
1) In the lecture it was said that in the implementation of bounded buffers using semaphores (see figure 1 on page 3) the order of the critSec.P() and noElts.P(), in process consumer, was essential, but that the order of critSec.V() noElts.V(), in process producer, was not. Identify the corresponding piece of code in the Buffer class provided and make the change. Can you produce an error situation? Note: You may see "error" messages about attempts to access a closed buffer. This is not the error you are looking for.

2) Why does the error situation arise when the code is changed as described in question 1? Why does it not arise in the original code?

3) Is the order of the calls of poll in the Buffer class's put method also essential?

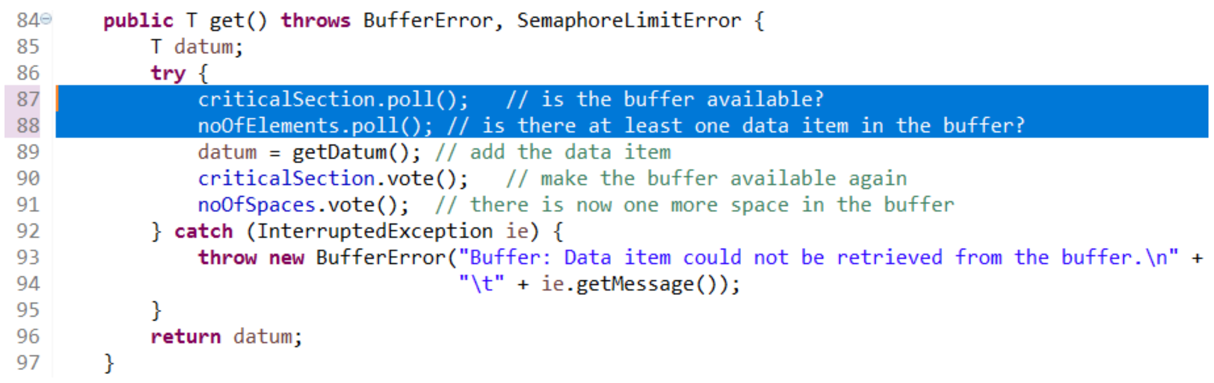
Answers:

1a) Identified code excerpt within the Buffer.java class:



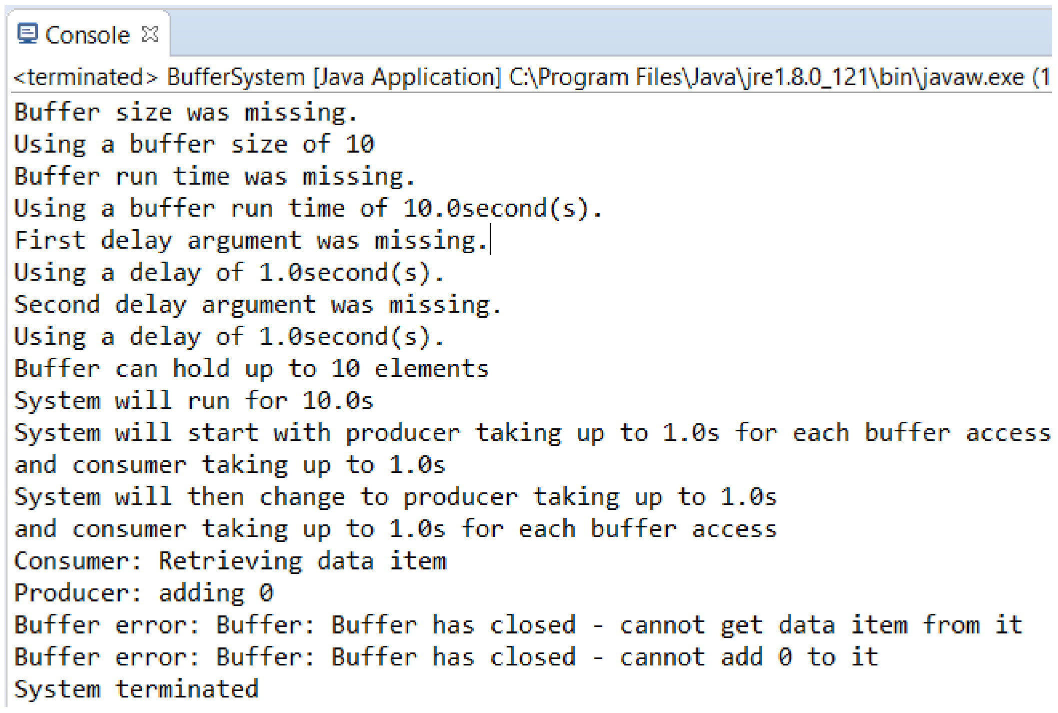
```java
    public T get() throws BufferError, SemaphoreLimitError {
        T datum;
        try {
            noOfElements.poll(); // is there at least one data item in the buffer?
            criticalSection.poll();   // is the buffer available?
            datum = getDatum(); // add the data item
            criticalSection.vote();   // make the buffer available again
            noOfSpaces.vote();  // there is now one more space in the buffer
        } catch (InterruptedException ie) {
            throw new BufferError("Buffer: Data item could not be retrieved from the buffer.\n" +
                                  "\t" + ie.getMessage());
        }
        return datum;
    }
```

1b) Changed order of critSec.P() and noElts.P() within code excerpt above:



```java
    public T get() throws BufferError, SemaphoreLimitError {
        T datum;
        try {
            criticalSection.poll();   // is the buffer available?
            noOfElements.poll(); // is there at least one data item in the buffer?
            datum = getDatum(); // add the data item
            criticalSection.vote();   // make the buffer available again
            noOfSpaces.vote();  // there is now one more space in the buffer
        } catch (InterruptedException ie) {
            throw new BufferError("Buffer: Data item could not be retrieved from the buffer.\n" +
                                  "\t" + ie.getMessage());
        }
        return datum;
    }
```

1c) Console output of error situation due to change in 1b):



```
<terminated> BufferSystem [Java Application] C:\Program Files\Java\jre1.8.0_121\bin\javaw.exe (1
Buffer size was missing.
Using a buffer size of 10
Buffer run time was missing.
Using a buffer run time of 10.0second(s).
First delay argument was missing.
Using a delay of 1.0second(s).
Second delay argument was missing.
Using a delay of 1.0second(s).
Buffer can hold up to 10 elements
System will run for 10.0s
System will start with producer taking up to 1.0s for each buffer access
and consumer taking up to 1.0s
System will then change to producer taking up to 1.0s
and consumer taking up to 1.0s for each buffer access
Consumer: Retrieving data item
Producer: adding 0
Buffer error: Buffer: Buffer has closed - cannot get data item from it
Buffer error: Buffer: Buffer has closed - cannot add 0 to it
System terminated
```

2) Why does the error situation arise when the code is changed as described in question 1? Why does it not arise in the original code?

In the modified code, the critical section is run before checking the number of elements in the buffer. This results in the consumer attempting to retrieve the data item before the producer can add to the buffer. This causes the error to occur, as the buffer will close, due to the consumer not being able to retrieve any data from the buffer. Also, because the buffer is now closed the producer cannot add to it. This will result in the system terminating.

In the original code, the number of elements in the buffer are checked before the critical section is run. This allows the producer to add to the buffer before the consumer retrieves the data item. Since there is now an element in the buffer, the consumer can retrieve it and the producer can continue adding to the buffer.

3) Is the order of the calls of poll in the Buffer class's put method also essential?

Yes. The order of the calls of poll in the Buffer class's put method are also essential because the process should confirm if there is any space in the buffer before the critical section is run.

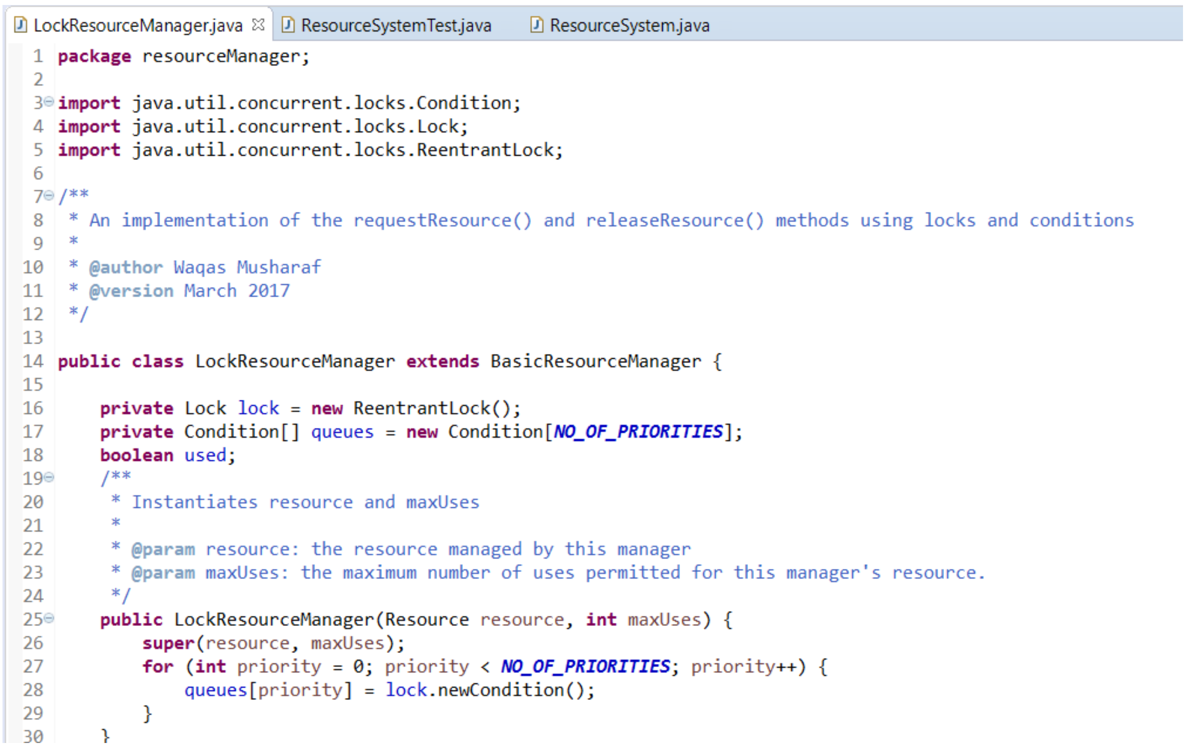
Tutorial Sixteen

Logbook Exercise:

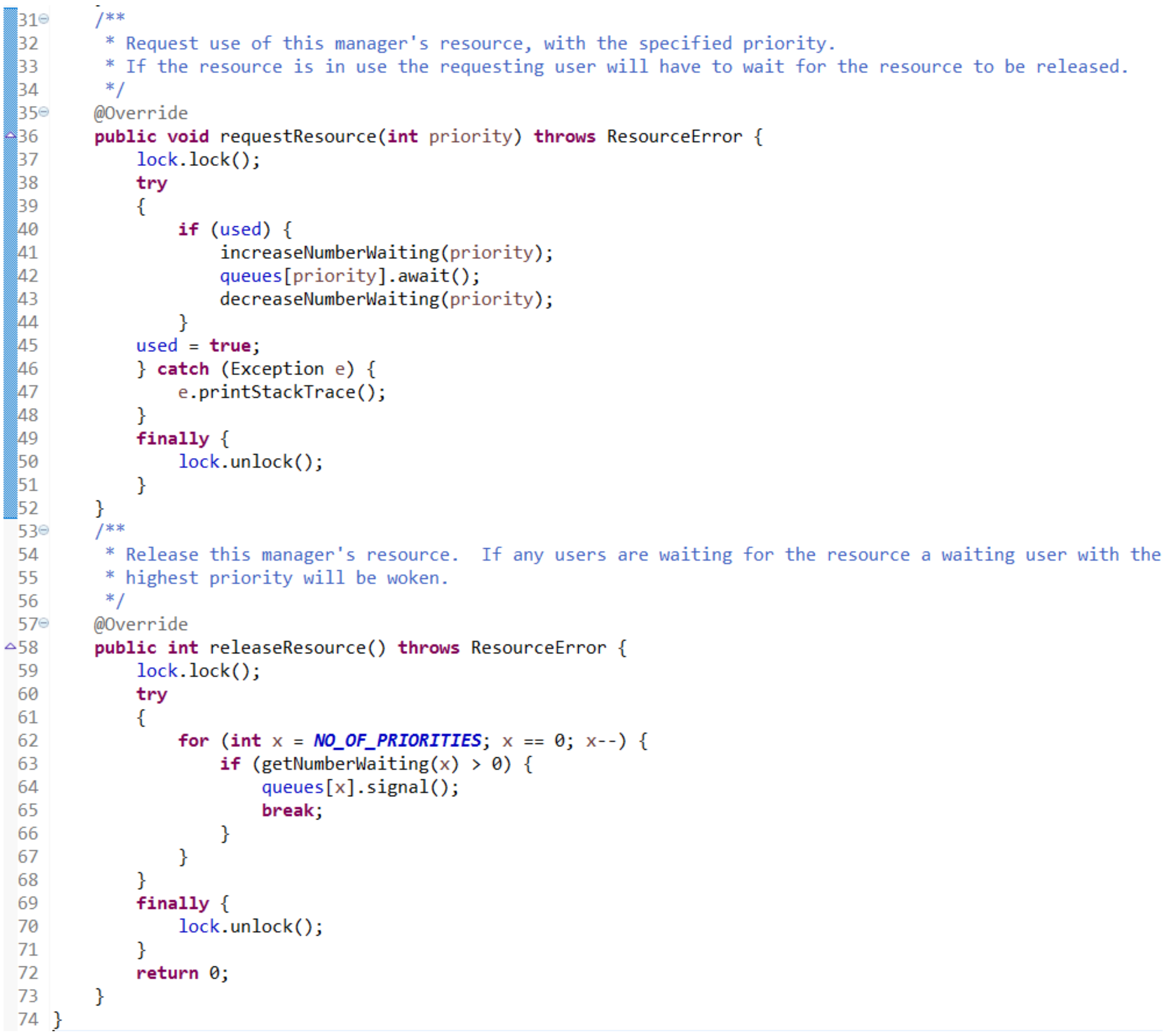
1) Implement your solution in Java, using Locks and Conditions. I.e., implement a LockResourceManager class that uses Locks and Conditions to implement the requestResource(int priority) and releaseResource() methods described above. Use the code provided (see section 3) to test your implementation, and try some tests of your own.

Answer:

1a) LockResourceManager.java class:

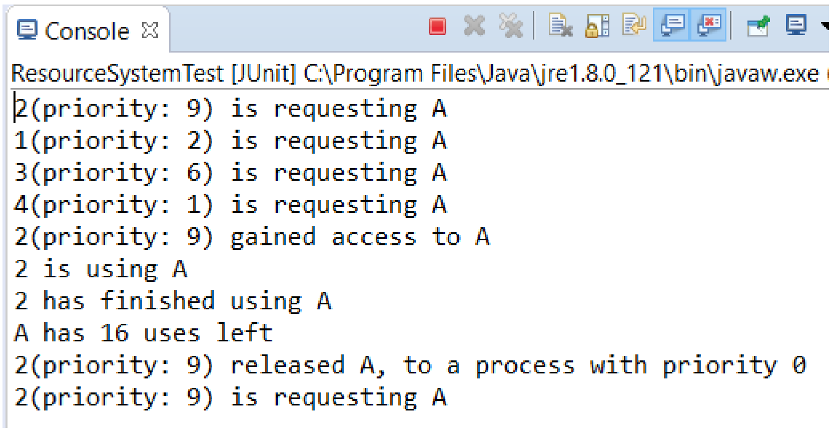


```java
package resourceManager;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * An implementation of the requestResource() and releaseResource() methods using locks and conditions
 *
 * @author Waqas Musharaf
 * @version March 2017
 */

public class LockResourceManager extends BasicResourceManager {

    private Lock lock = new ReentrantLock();
    private Condition[] queues = new Condition[NO_OF_PRIORITIES];
    boolean used;
    /**
     * Instantiates resource and maxUses
     *
     * @param resource: the resource managed by this manager
     * @param maxUses: the maximum number of uses permitted for this manager's resource.
     */
    public LockResourceManager(Resource resource, int maxUses) {
        super(resource, maxUses);
        for (int priority = 0; priority < NO_OF_PRIORITIES; priority++) {
            queues[priority] = lock.newCondition();
        }
    }
```



```java
    /**
     * Request use of this manager's resource, with the specified priority.
     * If the resource is in use the requesting user will have to wait for the resource to be released.
     */
    @Override
    public void requestResource(int priority) throws ResourceError {
        lock.lock();
        try
        {
            if (used) {
                increaseNumberWaiting(priority);
                queues[priority].await();
                decreaseNumberWaiting(priority);
            }
        used = true;
        } catch (Exception e) {
            e.printStackTrace();
        }
        finally {
            lock.unlock();
        }
    }
    /**
     * Release this manager's resource.  If any users are waiting for the resource a waiting user with the
     * highest priority will be woken.
     */
    @Override
    public int releaseResource() throws ResourceError {
        lock.lock();
        try
        {
            for (int x = NO_OF_PRIORITIES; x == 0; x--) {
                if (getNumberWaiting(x) > 0) {
                    queues[x].signal();
                    break;
                }
            }
        }
        finally {
            lock.unlock();
        }
        return 0;
    }
}
```

1b) Console output when running ResourceSystemTest.java (class provided), which creates an instance of the ResourceSystem.java class (class provided) which calls LockResourceManager.java:



```
ResourceSystemTest [JUnit] C:\Program Files\Java\jre1.8.0_121\bin\javaw.exe
2(priority: 9) is requesting A
1(priority: 2) is requesting A
3(priority: 6) is requesting A
4(priority: 1) is requesting A
2(priority: 9) gained access to A
2 is using A
2 has finished using A
A has 16 uses left
2(priority: 9) released A, to a process with priority 0
2(priority: 9) is requesting A
```

Although Week 17 was a regular week, with logbook exercises set (which can be seen later), there was also a self-assessment form completed during Week 17, which can be seen below:

Self-Assessment Three

Week(s)	Overall	Documentation	Structure	Naming	Testing	Functionality
13 Counter Behaviour	A	-	-	-	-	-
14 Dekker Trains	A	B	A+	A	A	A
15 Semaphore Behaviour	A	-	-	-	-	-
16 Locks and Conditions	B	B	A	A	C	B

Assessment Criterion	Grade
Answers to flagged logbook questions	A
Answers to other practical questions	C
Other practical work	C
Understanding of the module material to date	A/B
Level of self-reflection & evaluation	A+
Participation in timetabled activities	B
Time spent outside timetabled classes	B

Comments:

Week 13: I believe that I have explained counter behaviour well, as I explained why the test would always terminate, as well as gave reasonable explanation of theoretical minimum and maximum values. I also, where possible, provided screenshots to help me back up my explanation. Furthermore, I provided code of the test method I created and the JUnit test results of the method, to prove that it was working. Overall, I believe that I have answered this question well.

Week 14: I believe that the quality of documentation for this exercise was acceptable, however it could have been improved with additional comments within the code. The structure and naming was strong throughout the code I implemented. The code performed as expected when tested, a full run of which I provided screenshots for. For that reason, I believe that testing and functionality of the code were sufficient.

Week 15: I believe that I have explained semaphore behaviour well, as I explained why the error situation occurs in the modified code and not in the original code. I also provided screenshots of how I modified the code and the error that I received. Additionally, I explained why the order of the calls of poll in the Buffer class's put method is also essential. Overall, I believe that I have answered this question well.

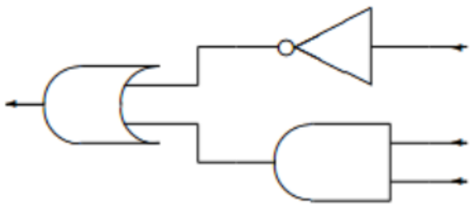
Week 16: I believe that the quality of documentation for this exercise was acceptable, however it could have been improved with additional comments within the code. The naming and structure were good throughout the code. Testing was lacking in this exercise though, and I plan to add some more testing later. The functionality of the code seems strong, but cannot be truly attested without substantial testing documentation.

Overall, I believe that I have answered the flagged logbook questions well, although additional work has been lacking. I should strive to add more additional work to the logbook as I continue with the module. So far, my understanding of this module's material is reasonable, although I feel like the exercises are increasing in difficulty as the module progresses. I believe my engagement with the module during lessons is satisfactory, as well as my engagement outside of lessons. I should strive to engage with the module even more by attending all my timetabled lessons and putting in more work outside of lessons.

Tutorial Seventeen

Logbook Exercise:

1) What is the matrix for the following circuit:



Answer:

1)

Firstly, I converted the circuit into a Boolean function which I calculated as:

¬ A ^ (B v C)

Following this, I created a truth table for this Boolean function, showing the output of the Boolean function according to all possible combinations of inputs. I then showed the outputs of the Boolean function in qubit form. Finally, I converted the qubit outputs into expanded matrix form:

A	0	0	0	0	1	1	1	1	**Inputs**
B	0	0	1	1	0	0	1	1	
C	0	1	0	1	0	1	0	1	
¬ A ^ (B v C)	1	1	1	1	0	0	0	1	**Output**
|¬ A ^ (B v C)>	|1>	|1>	|1>	|1>	|0>	|0>	|0>	|1>	**Matrix Output**
Expanded Matrix Form	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**0**	**Expanded Matrix Form**
	1	**1**	**1**	**1**	**0**	**0**	**0**	**1**	

What I calculated was the matrix:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

In order to confirm that this matrix was correct, I used MATLAB to calculate the same matrix again. In MATLAB, the matrices for the NOT gate and the AND gate were already implemented, but the matrix for the OR gate was not. Therefore, I created my own OR matrix by entering the following:

```
OR = [1,0,0,0;0,1,1,1];
```

After creating the OR matrix, I used the following function to get the matrix of the circuit:

```
>> OR*kron(NOT,AND)

ans =

     0     0     0     0     1     1     1     0
     1     1     1     1     0     0     0     1
```

MATLAB has confirmed that my tabular calculations were correct as the matrix generated in MATLAB was the exact same as the matrix I had derived from the given circuit.

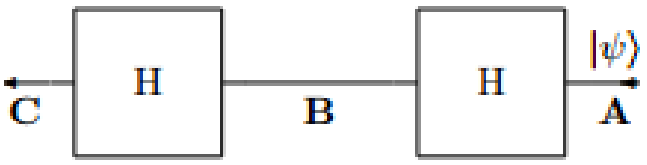
Due to Week 18 being a Guidance Week, there were no lecture or tutorial sessions and therefore no logbook exercises to complete. Also, Week 19 was a lecture and tutorial session on additional quantum computing, but there were no logbook exercises assigned.

Tutorial Twenty

Logbook Exercise:

1) If |ψ⟩ is a pure state (i.e. |ψ⟩ = |0⟩ or |ψ⟩ = |1⟩, what happens if |ψ⟩ is passed as input to a circuit consisting of two Hadamard gates in sequence.

Answer:



$$Hadamard = \left(\frac{1}{\sqrt{2}}\right) * \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$ZERO = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$ONE = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

1a) When |ψ⟩ = |0⟩

$$A = |\psi\rangle = |0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$B = Hadamard * ZERO = \begin{bmatrix} 0.7071 \\ 0.7071 \end{bmatrix}$$

$$C = Hadamard * \begin{bmatrix} 0.7071 \\ 0.7071 \end{bmatrix} = \begin{bmatrix} 1.0000 \\ -0.0000 \end{bmatrix}$$

1b) When |ψ⟩ = |1⟩

$$A = |\psi\rangle = |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$B = Hadamard * ONE = \begin{bmatrix} 0.7071 \\ -0.7071 \end{bmatrix}$$

$$C = Hadamard * \begin{bmatrix} 0.7071 \\ -0.7071 \end{bmatrix} = \begin{bmatrix} -0.0000 \\ 1.0000 \end{bmatrix}$$

A Hadamard gate, when applied to either a 0 or 1 qubit (as demonstrated in the exercise above), will produce a quantum state which, upon observation, will be a 0 or a 1 with equal probability. This is comparable to flipping a fair coin, where the result will be either heads or tails with equal probability.

When a Hadamard gate is applied in succession to the output of the product of another Hadamard gate with either a 0 or 1 qubit, the output from the second Hadamard gate is the exact same as the original input. This is similar to having constantly alternating coin faces every time you flip a fair coin.

This results in A and C being exactly the same as each other in both cases of $|\psi\rangle$, as applying two Hadamard gates in succession to A/C results in the output being exactly the same as the input.

Week 20 was the final week of assessment for this logbook. The logbook exercises have now been completed and no further logbook exercises will be documented.

Following Week 20, a final self-assessment was completed, which can be seen below:

Self-Assessment Four

Week(s)	Overall	Documentation	Structure	Naming	Testing	Functionality
17 Modelling Circuits	**A**	-	-	-	-	-
20 Quantum Computing	**A**	-	-	-	-	-

Assessment Criterion	Grade
Answers to flagged logbook questions	**A**
Answers to other practical questions	**C**
Other practical work	**C**
Understanding of the module material to date	**A**
Level of self-reflection & evaluation	**A+**
Participation in timetabled activities	**A**
Time spent outside timetabled classes	**A**

Week 17: I believe that I have answered this exercise well. I converted the circuit provided into matrix form and then calculated all possible matrix outputs from all possible inputs. I converted this data into qubit form and then into expanded matrix form upon which I derived my matrix. Additionally, I checked this matrix against the same calculation performed in MATLAB and received the same result. I think I explained my derivation well throughout the exercise, and I generally performed well throughout.

Week 20: I believe that I have also answered this exercise well. I fully analysed the values that appear at points A, B and C in the circuit. I also discussed the relationship between the values appearing at A and C. Furthermore, I explained the how this result compares to a real world probabilistic model. Overall, I believe I have performed well in this exercise.

Overall, throughout the entire logbook, I believe that I have generally answered flagged logbook questions to a high degree. Although lacking in documented additional work, I believe the logbook questions that I answered make up for the lack in other practical work. I believe that I understand the module well as a whole, although some aspects I may still find difficult. I have learnt a lot from creating this logbook and from the module as a whole. Overall, my engagement with the module has been high, as I have attended most of my timetabled sessions, and occasionally, some additional non-timetabled sessions. I have also spent a lot of time outside of lessons on this logbook and because of this, my understanding has increased further. I believe that the self-assessments completed regularly throughout the compilation of this logbook have been of good depth and helped me realise how I could improve in the future.

To conclude, I believe that this logbook is of a high standard: in content, self-assessment and general quality. This module has helped me learn a lot and helped me gain good study habits which I can apply to other assessments in the future.