CIS2344 Algorithms Processes and Data
Logbook

Part 1

Contents

Week 7

I implemented a solution using the List(T) Interface to create the following solution:

```
package linkedList.list;

import linkedList.node.ListNode;
import linkedList.node.SingleLinkNode;

/**
 * Allows nodes to form a Linked List
 * @param <T> the type of objects stored in the nodes.
 *
 * @author Adam Birch
 * @version November 2018
 */

public class SingleLinkedList<T> extends BasicList implements List {
    int nodeCount = 0;

    public SingleLinkedList() {
        this.root = new SingleLinkNode(null);
    }

    /**
     * Add a node to the list
     * @param index the index at which the new entry should be added.
     * @param value the value to be added.
     * @throws ListAccessError
     */
    @Override
    public void add(int index, Object value) throws ListAccessError {
        SingleLinkNode node = new SingleLinkNode(value);
        if (index < 0 || index > nodeCount){
            throw new ListAccessError("Requested index was outside of the list bounds.");
        }
        else if (index == 0) { // is Root
                if (nodeCount > 0){
                    SingleLinkNode nextNode = (SingleLinkNode) this.getRoot();
```

```
                this.setRoot(node);
                changeNext(index, nextNode);
            }
            else {this.setRoot(node);}
        }
        else if (index == nodeCount) { // is Tail
            addTail(node);
        }
        else {
            SingleLinkNode nextNode = (SingleLinkNode) this.get(index);
            node = new SingleLinkNode(value, nextNode);
            changeNext(index - 1, node);
            }
        nodeCount++;
    }

    /**
     * Adds a new node to the tail of the list.
     * @param node - The node to be added.
     * @throws ListAccessError
     */
    private void addTail(SingleLinkNode node) throws ListAccessError {
        SingleLinkNode oldTail = (SingleLinkNode) this.get(nodeCount-1);
        oldTail.setNext(node);
    }

    /**
     * responsible for changing the next value, for adding and deleting
     * @param index - The index of the node to be altered
     * @param nextNode - The new Next node
     */
        private void changeNext(int index, SingleLinkNode nextNode) {
        try {
            SingleLinkNode node = (SingleLinkNode) this.get(index);
            node.setNext(nextNode);
        } catch (ListAccessError listAccessError) {
            listAccessError.printStackTrace();
        }
    }

    /**
     * A method that takes only an object value as the new nodeCount node.
     * @param value - The value to be used as the new node.
     * @throws ListAccessError
     */
    public void add(T value) throws ListAccessError {
        add(nodeCount, value);
    }

    /**
     * Remove a node from the linked list.
     * @param index the index of the entry to be removed.
     * @return the node removed
     * @throws ListAccessError
     */
    @Override
    public Object remove(int index) throws ListAccessError {

        if (index > 1 && index < nodeCount){
            SingleLinkNode nextNode = (SingleLinkNode) this.get(index + 1);
            changeNext(index - 1, nextNode);
        }
        else if (index == 0){
            root = get(1);
            this.setRoot(root);
        }
        else if (index == nodeCount){
            SingleLinkNode node = (SingleLinkNode) this.get(index - 1);
            node.setNext(null);
        }
        nodeCount--;
        return this.get(index);
```

```
    }

    public void removeAll() throws ListAccessError {
        this.setRoot(null);
        this.nodeCount = 0;
    }


    /**
     * Retrieve the node at an index
     * @param index the index of the entry to be accessed.
     * @return the node at the index.
     * @throws ListAccessError
     */
    @Override
    public ListNode get(int index) throws ListAccessError {
        ListNode current = root;
        if (index > nodeCount){throw new ListAccessError("Requested index was outside of the
list bounds.");}
        else{
            boolean found = false;
            int i = 0;

            do {
                if (i == index){
                    found = true;
                }
                else{
                    i ++;
                    try {
                        current = current.getNext();
                    } catch (Exception e) {
                        break;
                    }
                }
            }
            while (!found && i <= nodeCount);
            if (!found){return null;}
        }
        return current;
    }

    /**
     * Output data about each node within the linked List.
     * @return all nodes within the linked List
     */
    public String allToString() {
        String data = "";
        for (int i = 0; i < nodeCount; i++) {
            SingleLinkNode temp = null;
            try {
                temp = (SingleLinkNode) this.get(i);
            } catch (ListAccessError listAccessError) {
                listAccessError.printStackTrace();
            }
            data = data + temp.getValue();
        }
        return data;
    }

    public String getString(int index) throws ListAccessError {
        SingleLinkNode node = (SingleLinkNode) this.get(index);
        String value = "" + node.getValue();
        return value;
    }
}
```

This is the test code for the solution.

```java
package linkedList.list;

import arrayGenerator.generator.ArrayGenerator;
import arrayGenerator.generator.IntegerArrayGenerator;
import arrayGenerator.generator.StringArrayGenerator;
import linkedList.node.SingleLinkNode;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.util.ArrayList;
import java.util.Arrays;

import static org.junit.jupiter.api.Assertions.*;

class SingleLinkedListTest {

    SingleLinkedList list = new SingleLinkedList();
    ArrayList expected = new ArrayList();

    /**
     * Create the String List.
     * @return NULL as List and Expected are already defined
     * @override List with new values.
     * @throws ListAccessError
     */
    void setUpString() throws ListAccessError {
        StringArrayGenerator strGenerator = new StringArrayGenerator();
        list = new SingleLinkedList();
        expected = new ArrayList();
        String[] temp = strGenerator.getArray(10);
        for (int i = 0; i < temp.length; i++){
            list.add(temp[i]);
            expected.add(temp[i]);
        }
    }

    /**
     * Create the Integer List.
     * @return NULL as List and Expected are already defined
     * @override List with new values.
     * @throws ListAccessError
     */
    void setUpInteger() throws ListAccessError {
        IntegerArrayGenerator intGenerator = new IntegerArrayGenerator();
        list = new SingleLinkedList();
        expected = new ArrayList();
        Integer[] temp = intGenerator.getArray(10);
        for (int i = 0; i < temp.length; i++){
            list.add(temp[i]);
            expected.add(temp[i]);
        }
    }

       /**
     * Test that the Get function will return the expected value.
     *
     * @throws ListAccessError
     */

    @Test
    void getFithStringtest() throws ListAccessError {
        setUpString();
        assertEquals("" + expected.get(5), list.getString(5));
    }

    /**
     * Test that a value can be added and that it is added to the correct position.
     * @return True if 'a' is added to position 5 and is stored correctly.
     * @throws ListAccessError
     */
    @Test
    void addTo5Test() throws ListAccessError {
```

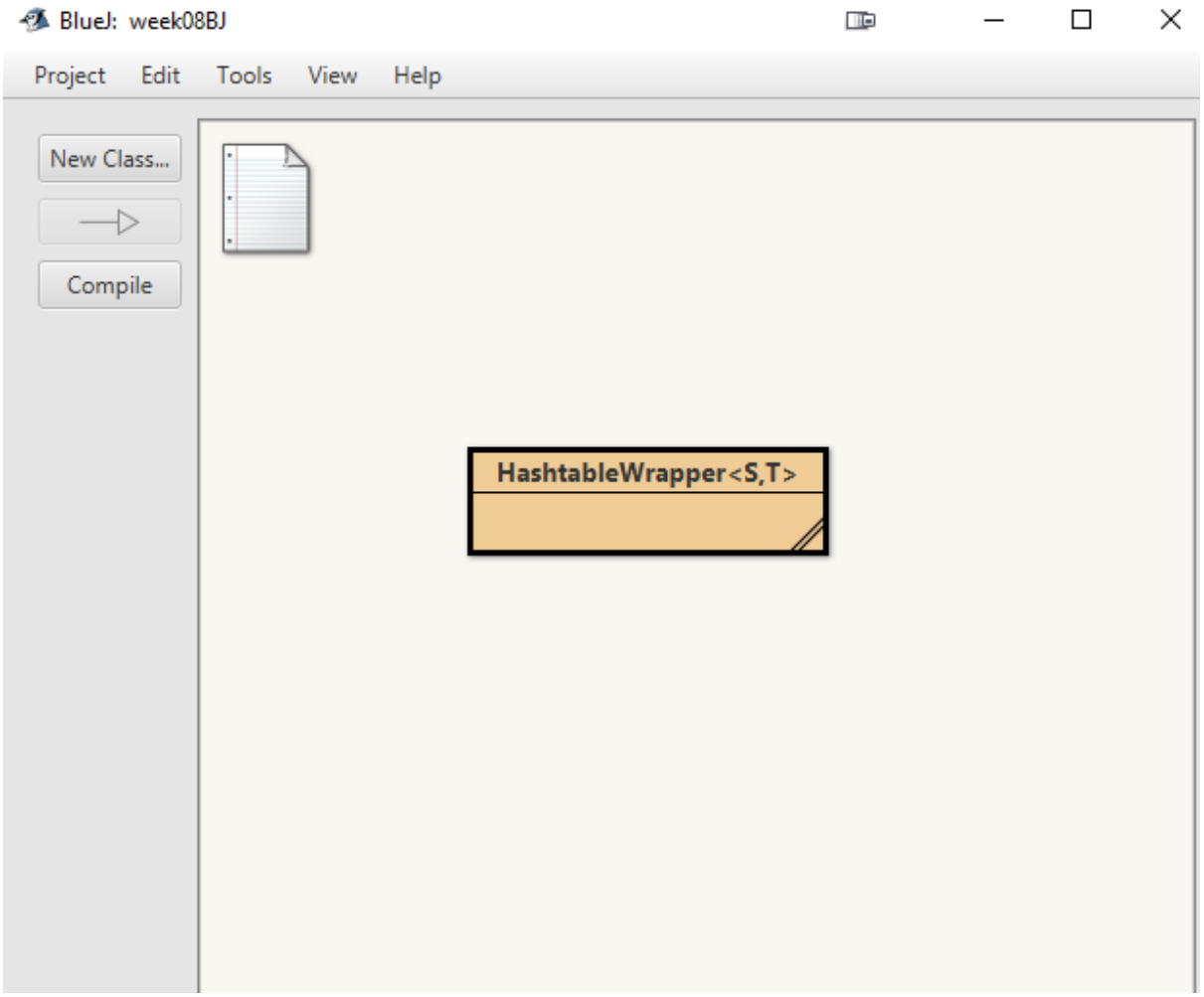
```
        setUpString();
        list.add(5, 'a');
        assertEquals("a", list.getString(5) );
    }

    /**
     * Test to check that the list works with Integer values.
     * This runs the same test as the getFifthStringTest.
     * @return True if the 5th Integer is correctly identified.
     * @throws ListAccessError
     */
    @Test
    void getFithIntegerTest() throws ListAccessError {
        setUpInteger();
        assertEquals("" + expected.get(5), list.getString(5));
    }

    /**
     * Test that a ListAccessError is thrown for Integers.
     * @return True if the error is thrown
     * @throws ListAccessError as 12 is out of bounds.
     */
    @Test
    void getOutOfBoundIntegerTest() throws ListAccessError {
        setUpInteger();
        assertThrows(ListAccessError.class, ()->{list.get(12);}); // 12 is out of bounds
    }

    /**
     * Test that a ListAccessError is thrown for Strings.
     * @return True if the error is thrown
     * @throws ListAccessError as 12 is out of bounds.
     */
    @Test
    void getOutOfBoundStringTest() throws ListAccessError {
        setUpString();
        assertThrows(ListAccessError.class, ()->{list.get(12);}); // 12 is out of bounds
    }
}
```

Self-Assessment

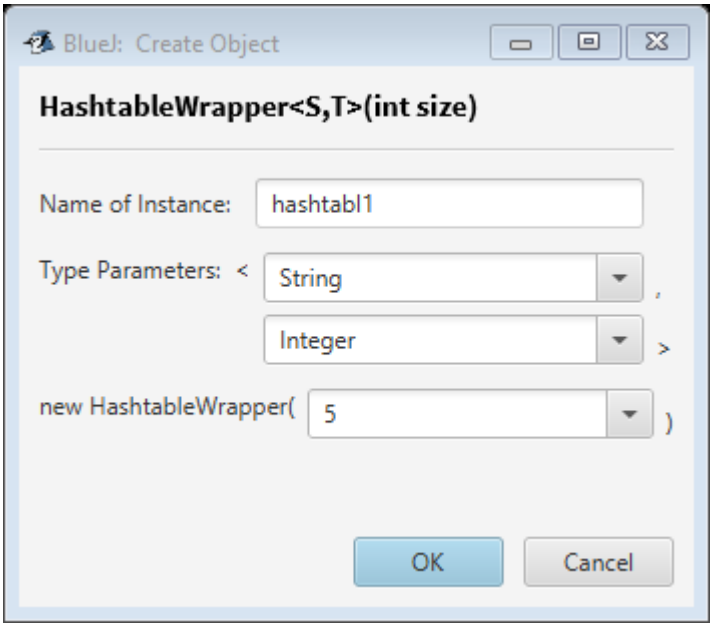
Rating - 4

I would give myself 4/5 for this week as I was able to create a working solution to the problem but I don't feel that the tests I have completed are thorough enough to warrant more marks than that. I was able to check for some exceptions within the solution and different types for the test (String and Integers).

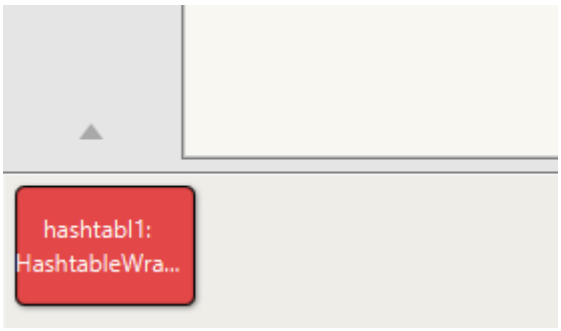
Week 8



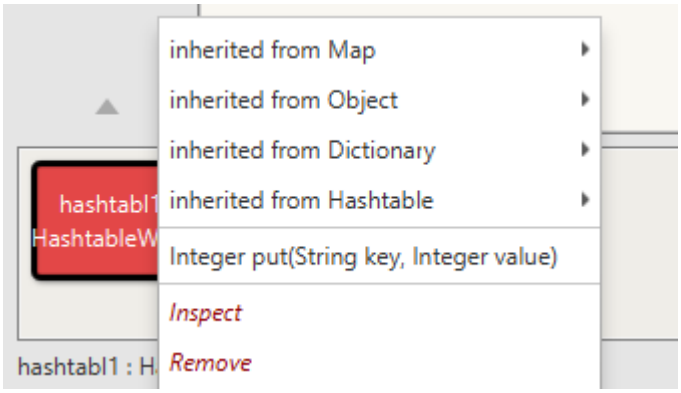
Within BlueJ I have Opened the Week8 project and am ready to start manipulating the object.



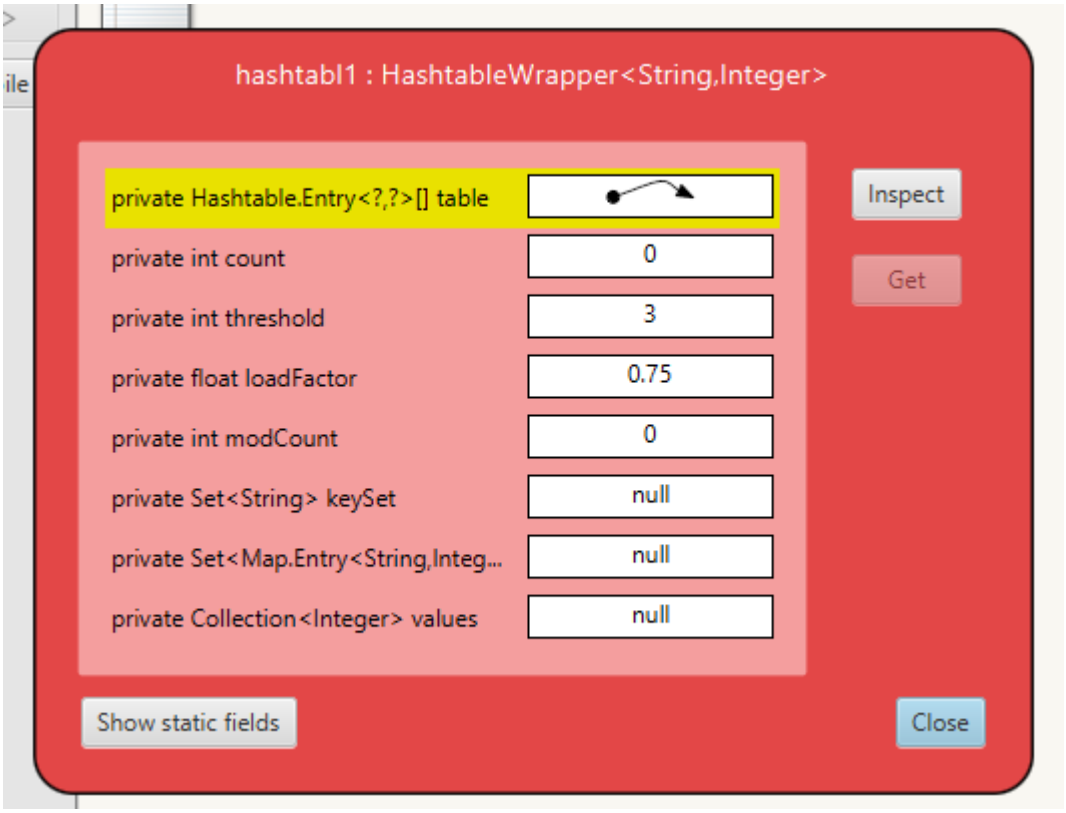
Within the creation window, I have specified the parameter types for parameters 1 and 2 as String and Integer respectively and set the initial size to 5. These types can be altered based on the use case of this object within a larger system, or have a different initial size of the array.



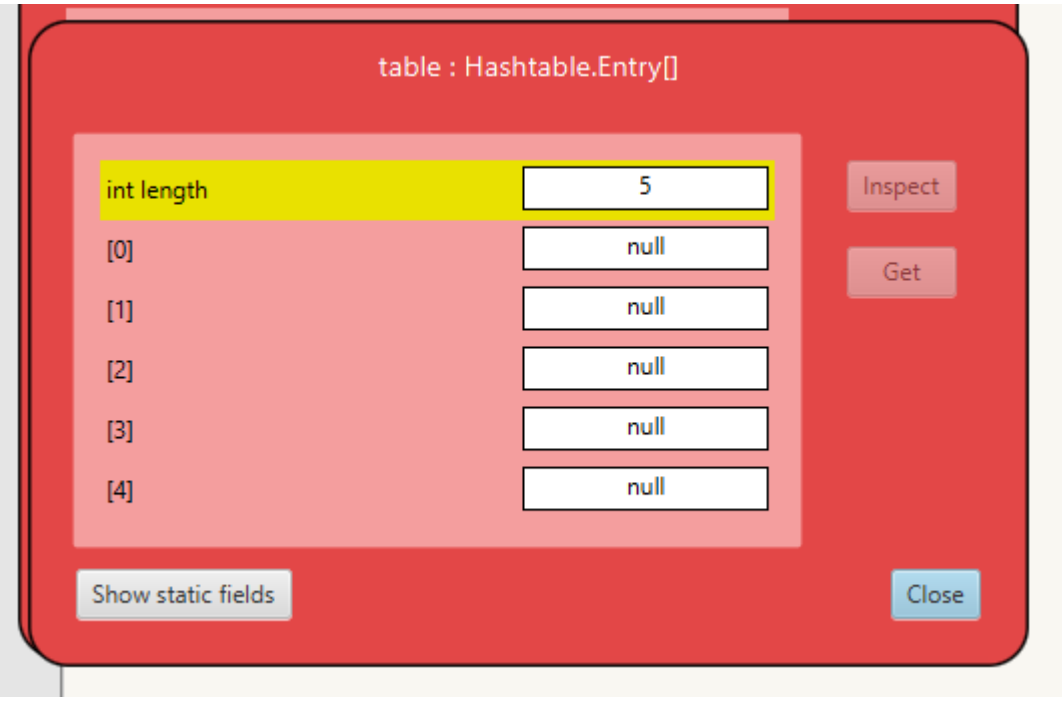
This new object (hashtabl1) has appeared at the bottom of the screen with the properties that were set in the previous step.



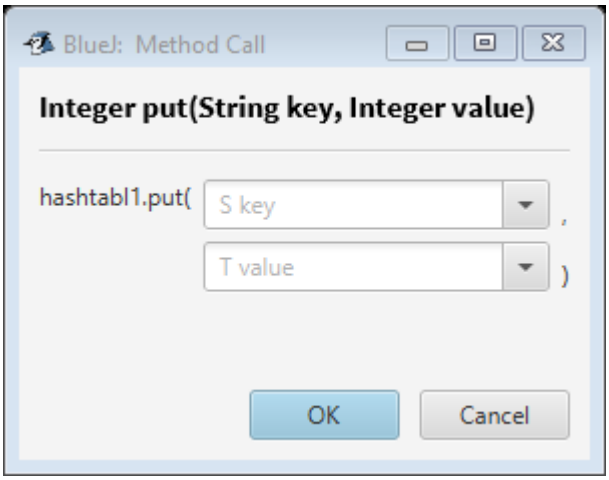
From the drop-down menu, I am able to inspect the contents of the object. As I have not set any data the object should be populated with the default values.



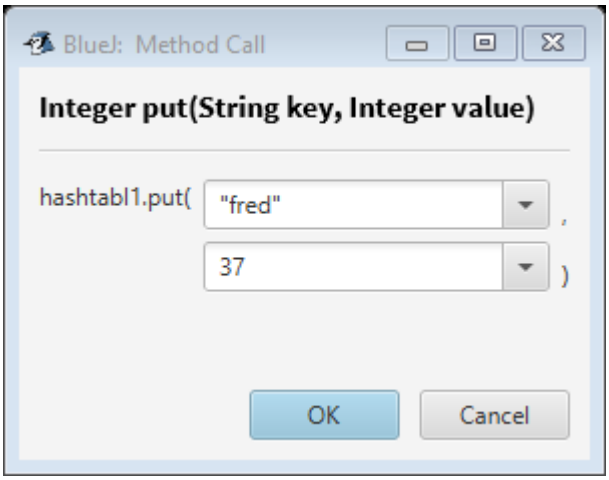
This object window shows detailed information about the contents of the object. There is no data within it, hence int Count and int modCount are both 0. There are empty sets and collections within the object, hence the last 3 fields are populated with null values. The object does point to the location of an array. This is to be expected as the array was initialised with a size of 5, so there is an array there – it just doesn't have any data yet.



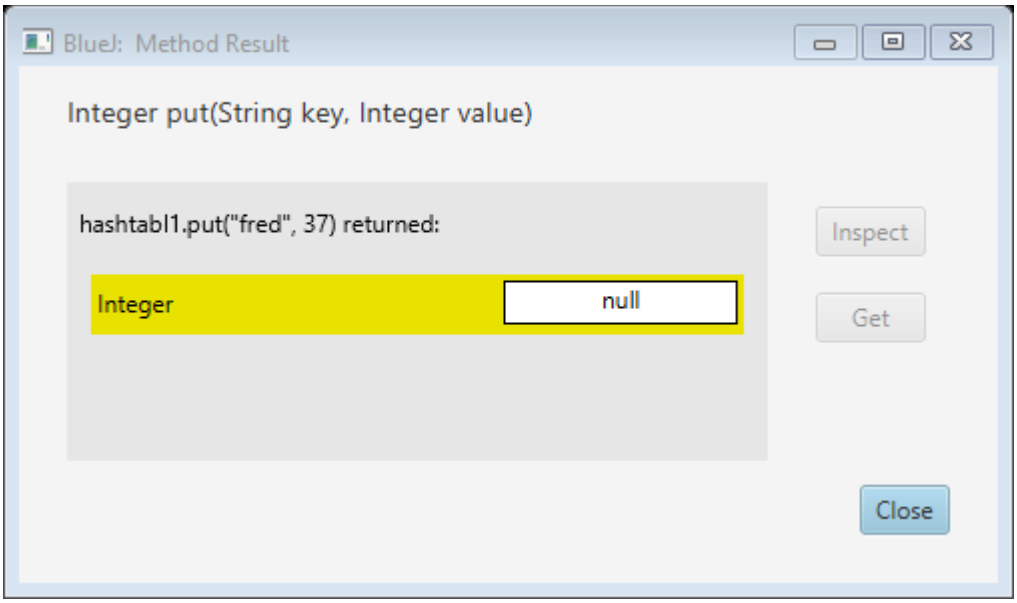
Upon inspecting the object, the internal array is populated with null values of the size 5 that was pre-defined when the object was created. The 5 locations are populated with null values as having an empty string is still data that can be extracted from the array, and null values are standard if no data was stored.



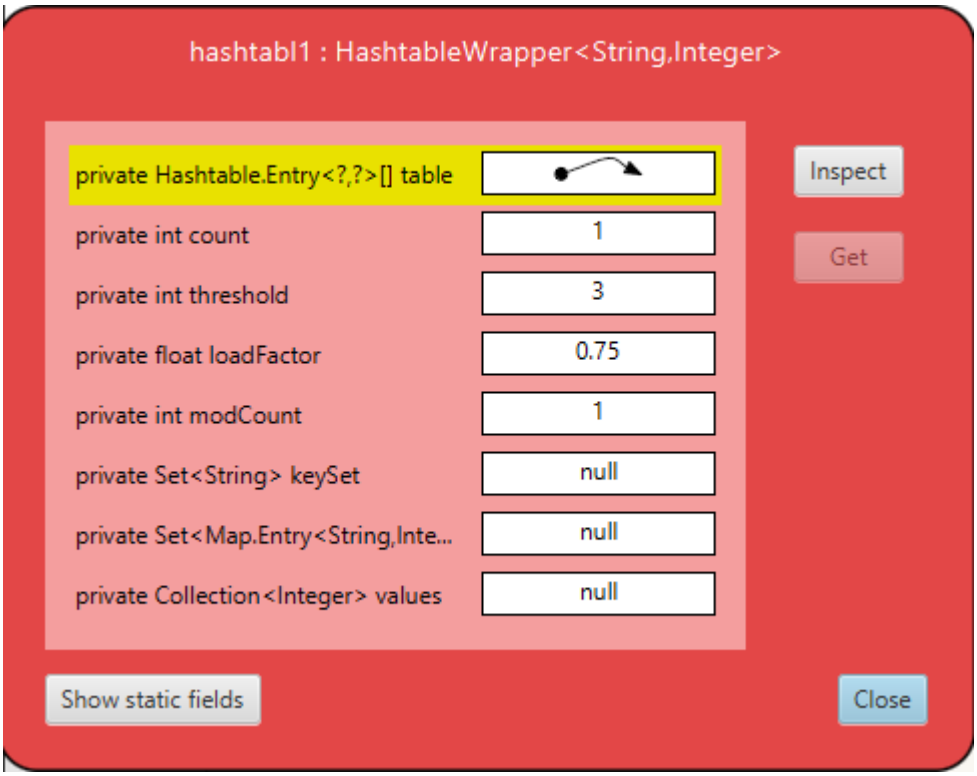
From the drop-down menu from before, using the put(String key, Integer value) method this window appears. I can now enter any compatible data and store it within the object.



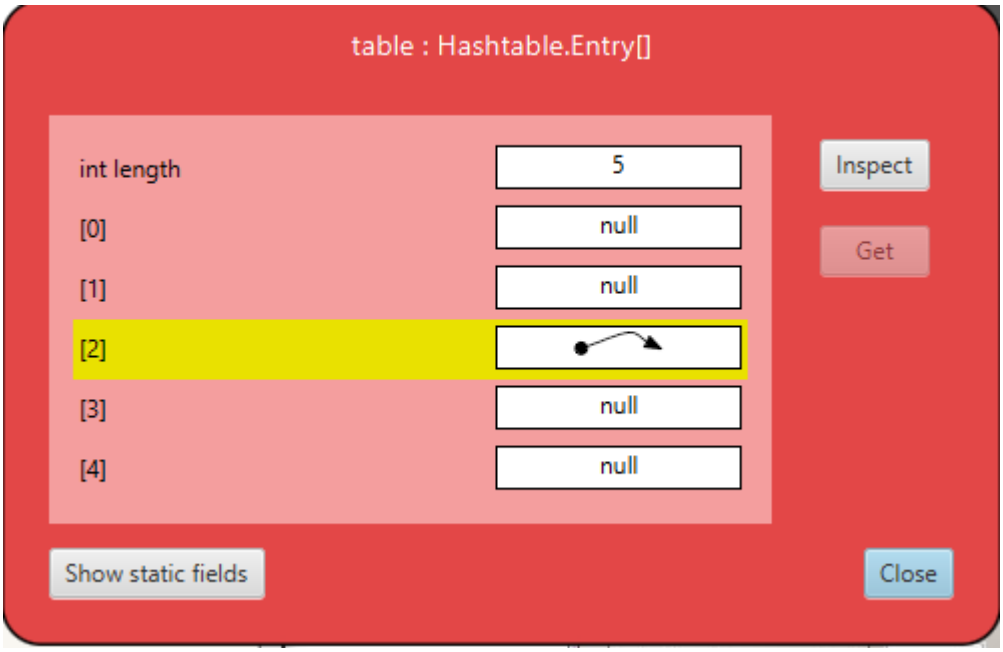
This is the populated input window. I will use this again later to fill in the rest of the data. String values must be encapsulated with quotation “” marks to define them as a string as this helps the computer to define a string from an int if numerical characters are entered. Int values must be whole numerical inputs as otherwise it will throw an exception.



As this window appears once the method is called, the input was a success. We can check this by looking again at the internal array of the object using the previous steps.

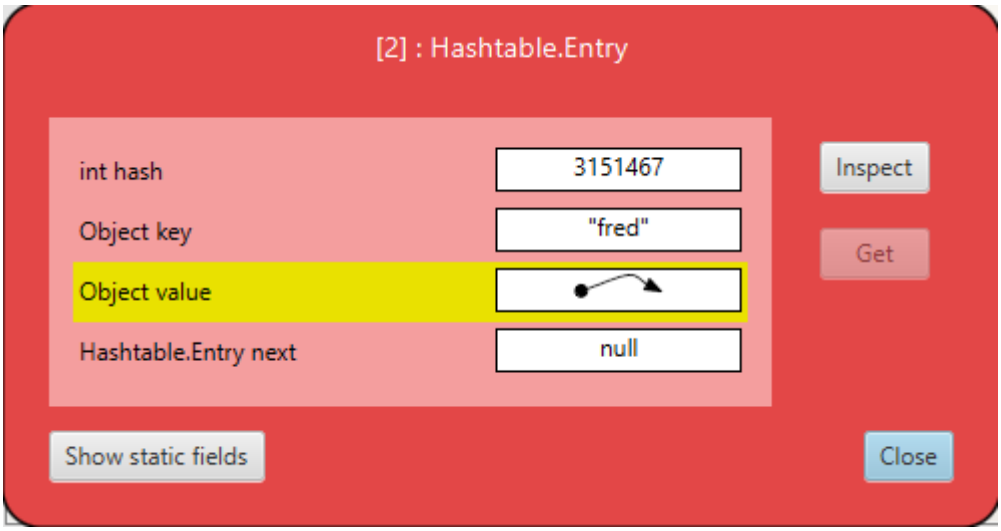


This is a view of the object. As there is now an int count and modCount of 1, there has been a change since looking at the object last. This should be the insert of “fred” that we did previously.

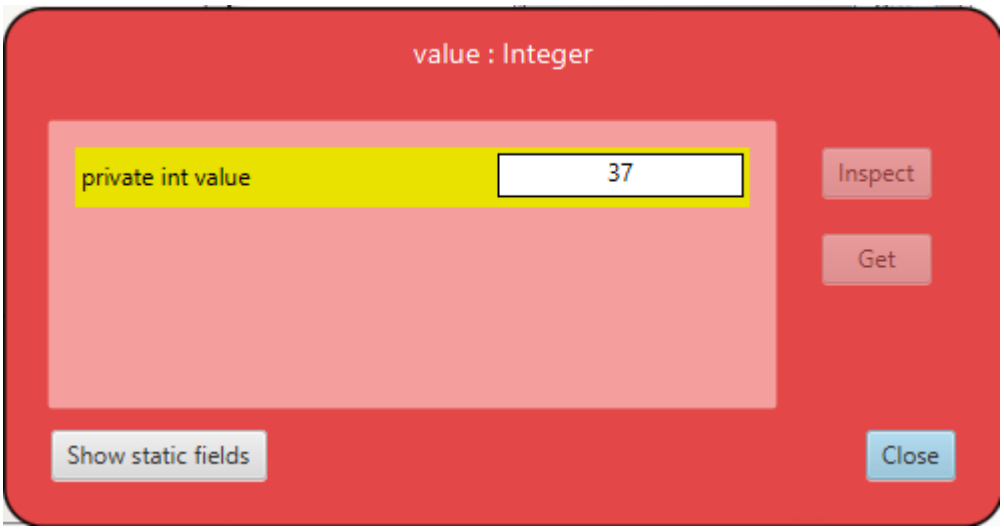


Inspecting the internal array presents this view. This shows that data has been entered into index 2.

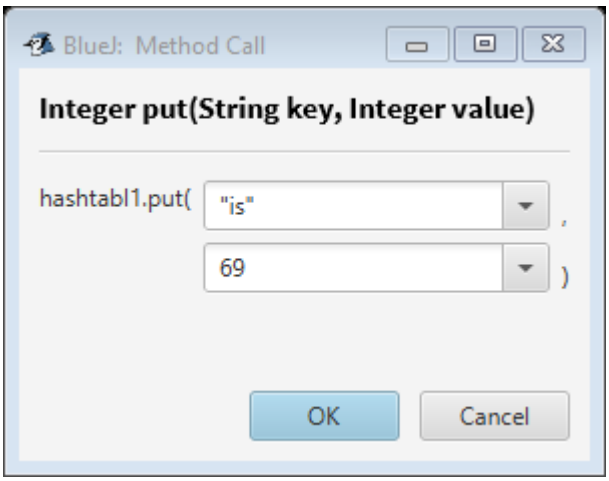
Why 2 and not 0? A key is converted into an integer by using a hash function. This integer can be used as an index to store the original element, which is then used for the position of the data within the hash table. In this case, “fred” was converted into a hash that was allocated to location 2 for the table.

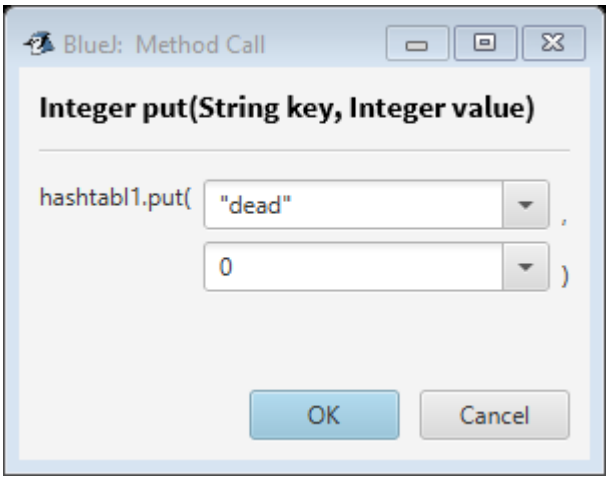


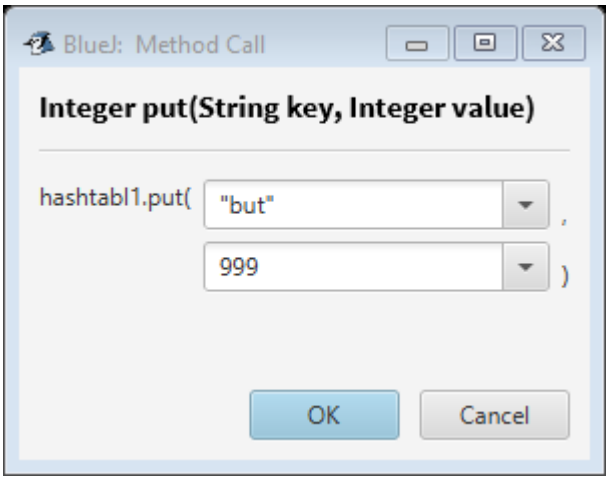
Inspecting the data at index 2 shows that “fred” has been entered with its own hash value, a link to it’s value, and has no next data (as no more data is present within this entry of the hash table). If there was a value for the next entry then that would suggest that there was a collision and multiple sets of data were stored at the same position in a linked list.

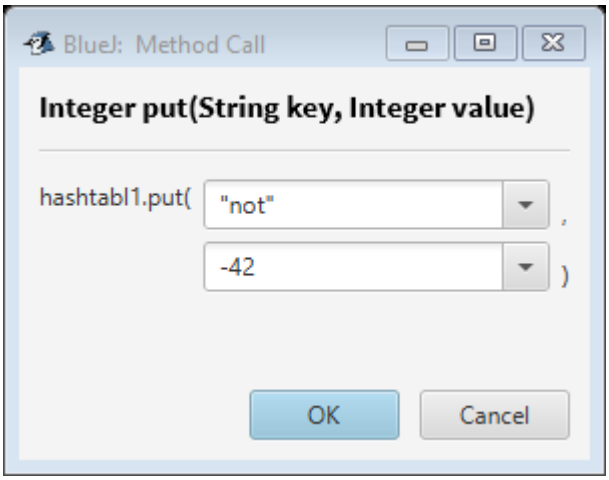


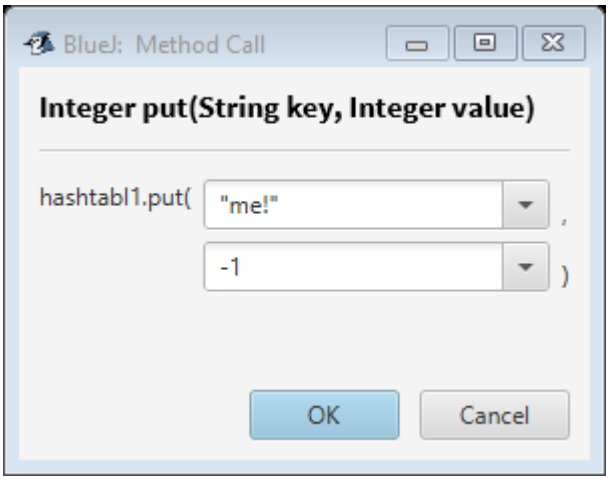
Inspecting the value of the object shows that the correct value was stored and that the method call was a success. This means that storing a key value pair is possible within this solution and that this implementation would work on a larger scale.



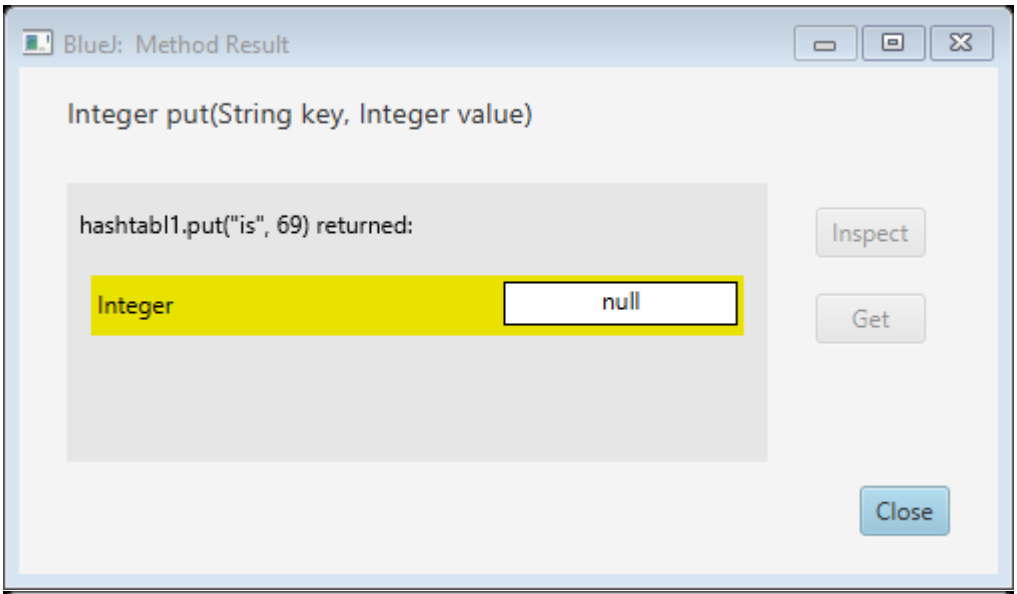


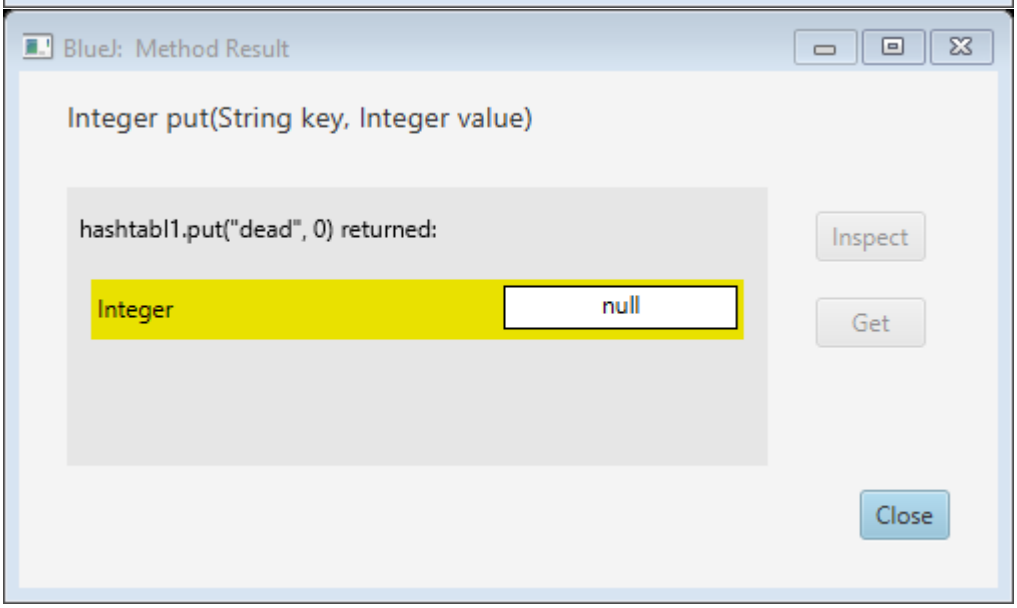


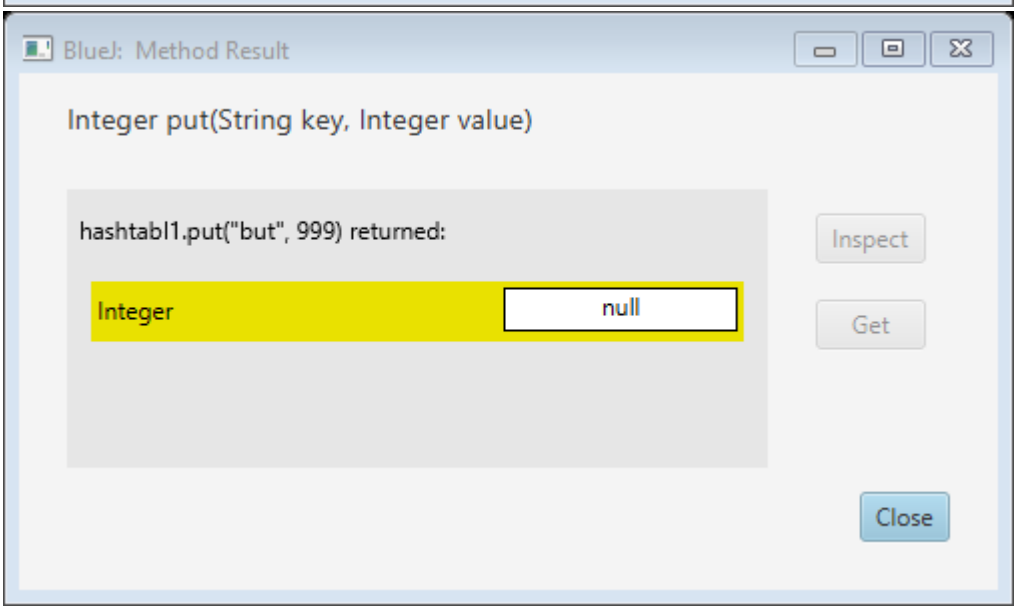


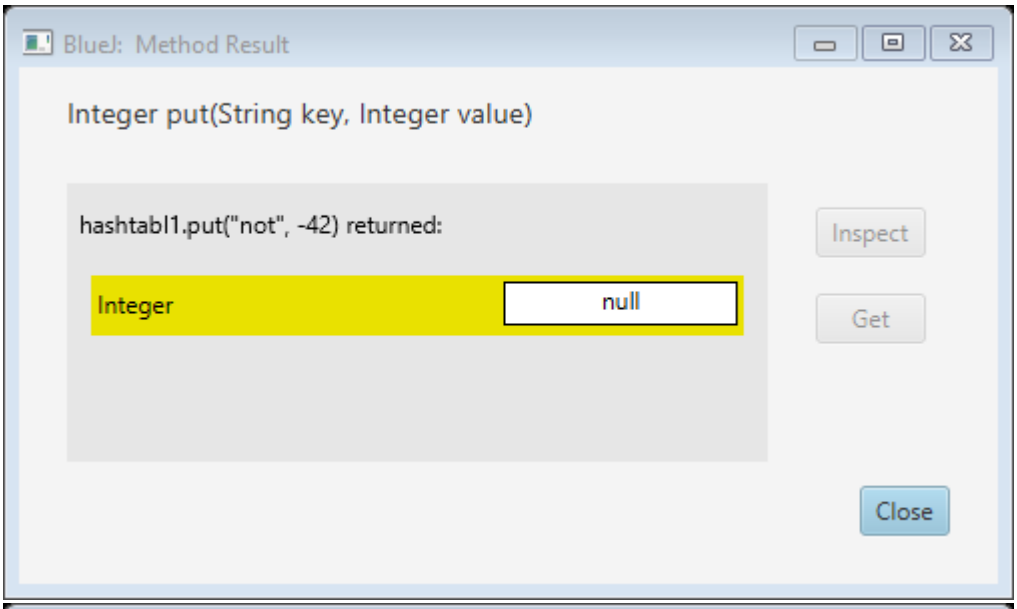


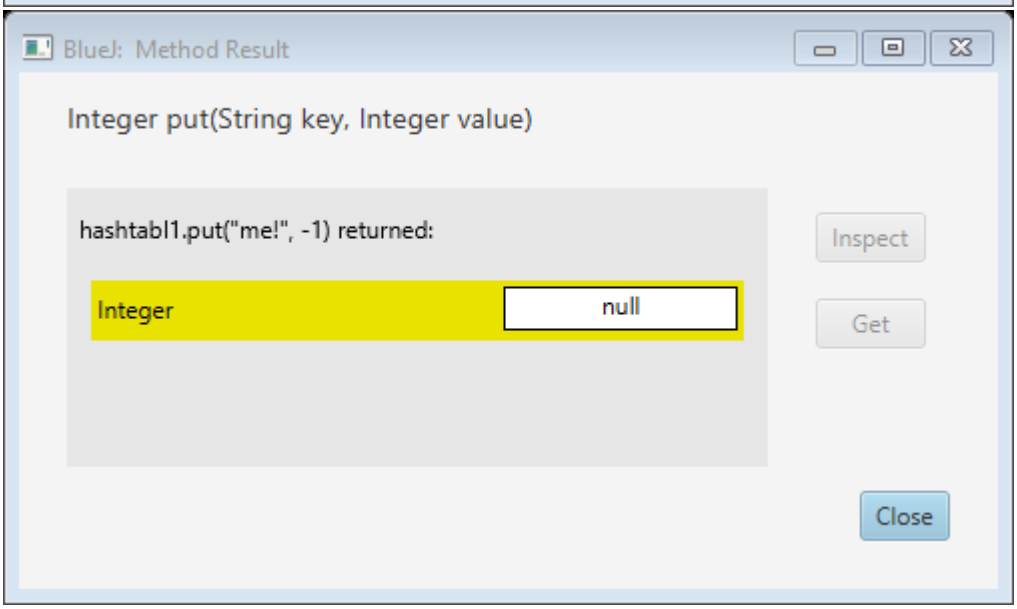
I have now entered all the above data into the object. These should be stored in separate locations within the hash table as each of the keys are distinct – and be less likely to collide with a well-written hash function.


BlueJ: Method Result
Integer put(String key, Integer value)
hashtabl1.put("is", 69) returned:
Integer
null
Inspect
Get
Close

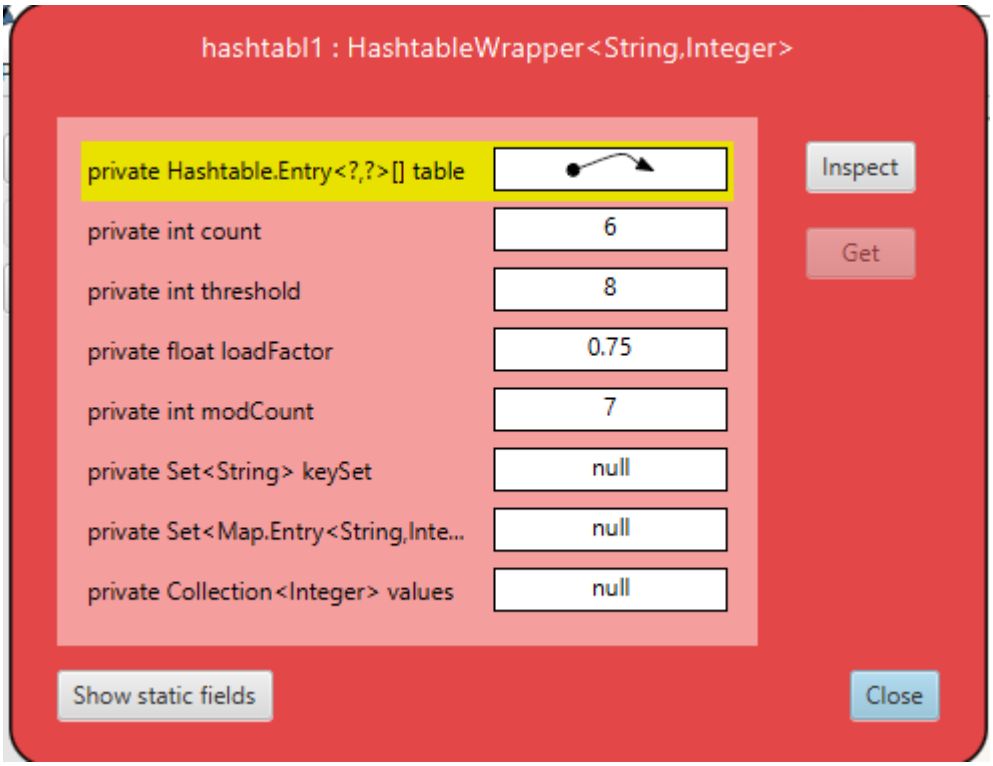

BlueJ: Method Result
Integer put(String key, Integer value)
hashtabl1.put("dead", 0) returned:
Integer
null
Inspect
Get
Close


BlueJ: Method Result
Integer put(String key, Integer value)
hashtabl1.put("but", 999) returned:
Integer
null
Inspect
Get
Close

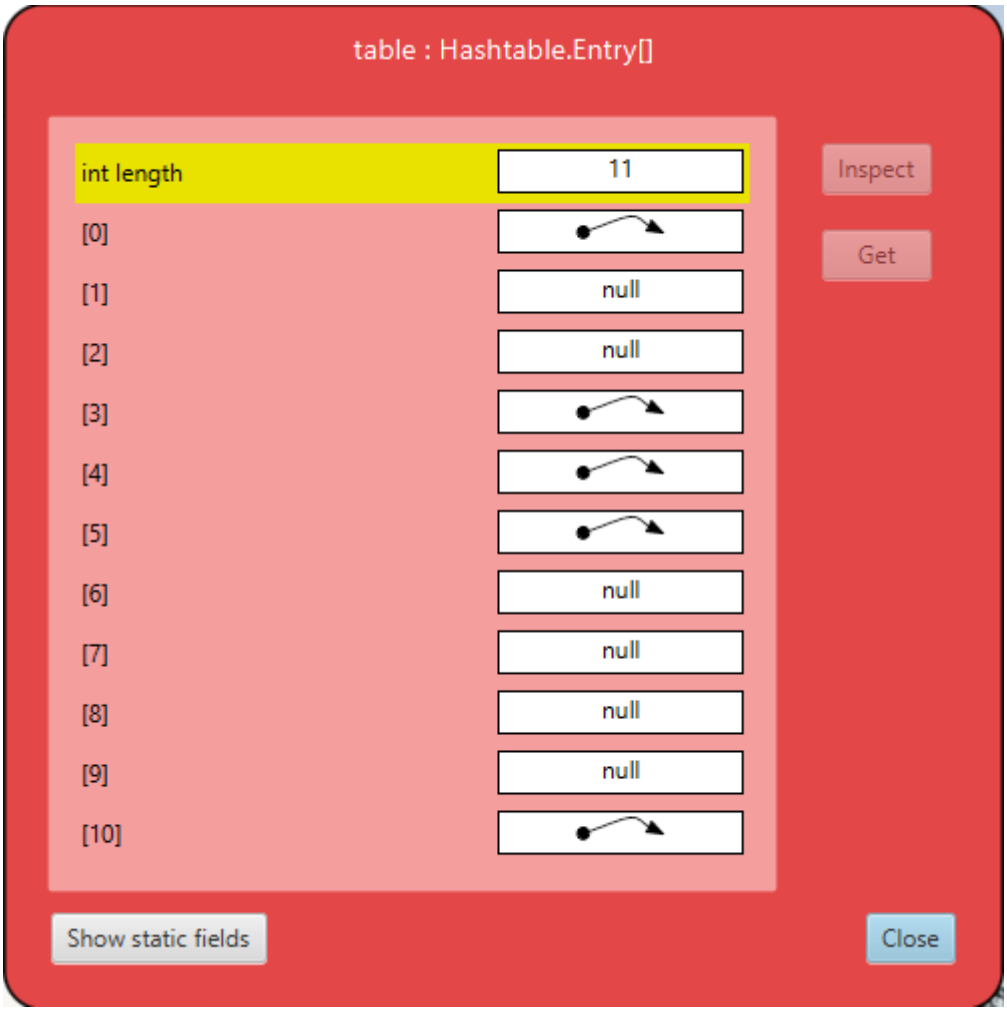




All the outputs suggest that the data was successfully added to the object. This can be verified by looking at the contents of the hash table and checking that each key and value are correctly stored – even if the order in which they were entered was lost.

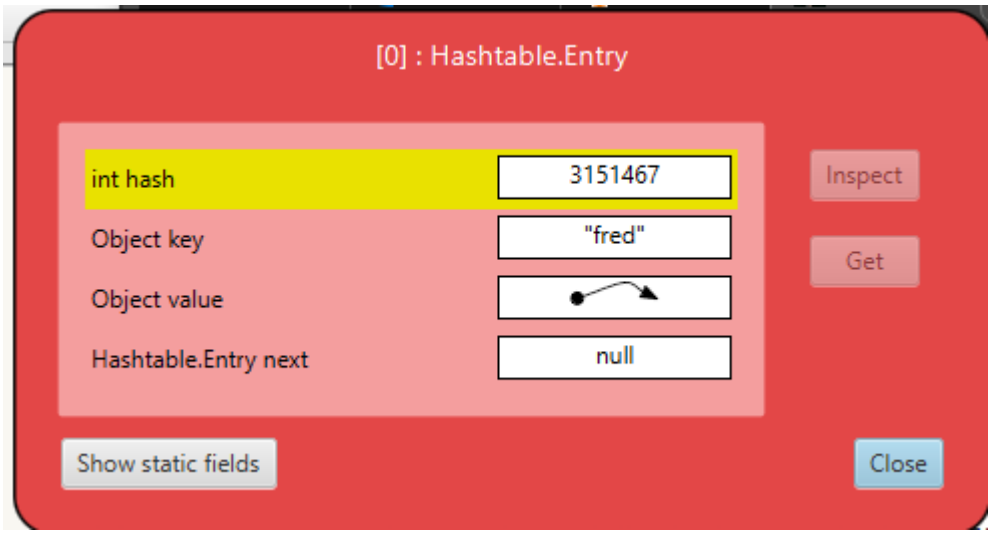


This is a view of the object after adding all the data. We have now come across a discrepancy – int count shows 6, while int modCount shows 7. This means there was an extra modification made to the object to store this data.

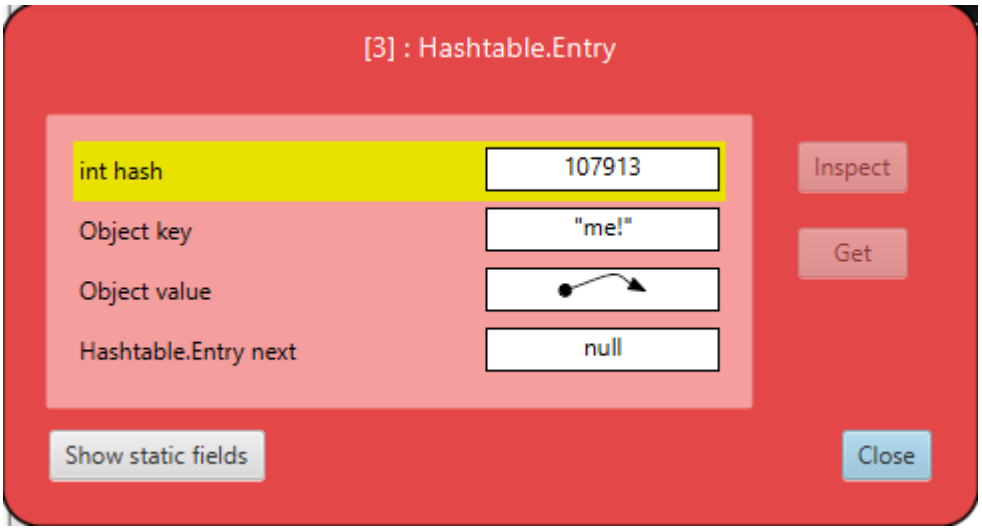


With this view of the internal array, we can see that the extra modification was to increase the length of the array, but not from 5 to 6, but from 6 to 11. This is one of the problems with this sort of file storage, as it is not always

possible to add data into the slots, there can be many wasted memory locations within the hash table. On a small scale like this it isn't a problem however on larger systems with millions or billions of entries, this can waste gigabytes or terabytes of data with null values.



"fred" appears first within the array. This is due to the hash value changing from position 2 to position 0 as the size of the hash table increased.



The next data appears in location 3 and is "me!", the last data to be inserted.

The nature of Hash functions means that the order of data inserted is not stored (natively) as similar data may eventually clash with data already in the table. E.G. if index 5 and 6 were Cat and Cut and the next data to be inserted is Cot, the data may clash as it should go between the two sets of data alphabetically, or after them based on the order – depending on the hash function used. However, the Hash values would be different and can be used to generate the location that the data should be stored in.

This Hash function that was used for this object is effective at eliminating the chances of collisions – but has created wasted space within the data. 6/11 elements within the array are null – over half of the size of the array. This can be filled with data providing that more data is added and that those spaces are correct for the data to be inserted. However it is just as likely that the size of the array will be increased, meaning that a new array will be created, the data copied over, and the original deleted – taking up more processing power than is required and is noticeable by a

user with very large sets of data. To create a hash function that creates little gaps in the table yet is quick to compute and results in few collisions is complicated and time consuming.

Self-Assessment

Rating - 4

I would give myself 4/5 for this weeks work as I feel that I have fully explained the issues surrounding memory usage and explained how a hash value is created and stored based on the key provided and the hash function used. However, there are likely many elements of hash algorithms that I omitted from this which is why I can't justify a 5/5.

Week 9

This is the BinaryTree class used for the implementation of the solution.

```
package binaryTree;

import java.util.ArrayList;
import java.util.List;

/**
 * An implementation of the Binary Tree work, extending class BTree
 * @author Adam Birch
 * @version December 2018
 */

public class BinaryTree<T extends Comparable<? super T>> implements BTree<T> {

    private static List traversalList = new ArrayList();
    private static int nodeCount = 0;

    private TreeNode<T> root; // the root node
    private BTree<T> left, right;

    @Override
    public String toString() {
        return "" + root;
    }

    /**
     * Construct an empty tree.
     */
    public BinaryTree() {
        root = null;
    }

    /**
     * Construct a singleton tree.
     * A singleton tree contains a value in the root, but the left and right subtrees are
     * empty.
     * @param value the value to be stored in the tree.
     */
    public BinaryTree(T value) {
        root = new TreeNode(value);
    }

    /**
     * Construct a tree with a root value, and left and right subtrees.
     * @param value the value to be stored in the root of the tree.
     * @param left the tree's left subtree.
     * @param right the tree's right subtree.
     */
```

```
    public BinaryTree(T value,BinaryTree<T> left,BinaryTree<T> right) {
        root = new TreeNode(value,left,right);
    }

    /**
     * Check if the tree is empty.
     * @return true if the tree is empty.
     */
    @Override
    public boolean isEmpty() {
        return root == null;
    }

    /**
     * Insert a new value in the binary tree at a position determined by the current contents
     * of the tree, and by the ordering on the type T.
     * @param value the value to be inserted into the tree.
     */
    @Override
    public void insert(T value) {
        if(isEmpty()){
            root = new TreeNode(value);
            nodeCount ++;
        }
        else{
            if (value.compareTo(root.value) <= 0){
                    BTree<T> tree = getLeft();
                    tree.insert(value);
            }

            else if (value.compareTo(root.value) > 0){
                    BTree<T> tree = getRight();
                    tree.insert(value);
            }
        }
    }

    /**
     * Get the value stored at the root of the tree.
     * @return the value stored at the root of the tree.
     */
    @Override
    public T getValue() throws NullPointerException {

        return root.getValue();
    }

    /**
     * Change the value stored at the root of the tree.
     * @param value the new value to be stored at the root of the tree.
     */
    @Override
    public void setValue(T value) {
        // implement setValue(T value) here
        root.setValue(value);
    }

    /**
     * Get the left subtree of this tree.
     * @return  This tree's left subtree.
     */
    @Override
    public BTree<T> getLeft() throws NullPointerException {
        // placeholder return value below - replace with implementation of getLeft()
        return root.left;
    }

    /**
     * Change the left subtree of this tree.
     * @param tree the new left subtree.
     */
    @Override
```

```java
    public void setLeft(BTree<T> tree) {
        // implement setLeft(BTree<T> tree) here
        root.left = tree;
    }

    /**
     * Get the right subtree of this tree.
     * @return this tree's right subtree.
     */
    @Override
    public BTree<T> getRight() throws NullPointerException{
        // placeholder return value below - replace with implementation of getRight()
        return root.right;
    }

    /*
     * Change the right subtree of this tree.
     * @param tree the new right subtree.
     */
    @Override
    public void setRight(BTree<T> tree) {
        // implement setRight(BTree<T> tree) here
        root.right = tree;
    }

    /**
     * Check if the tree contains a given value.
     * @using Preorder traversal.
     * @param target the value to be checked.
     * @return true iff the value is in the tree.
     */
    @Override
    public boolean contains(T target) {
        T rootValue = null;
        try {
            rootValue = this.root.getValue();
        } catch (NullPointerException e) { // null = value not found
            return false;
        }
        boolean found = false;
         if (target.compareTo(rootValue) == 0){found = true;}
        else{
            if (rootValue.compareTo(target) > 0){ // If target is smaller than the root
                BTree next = root.getLeft();
                found = next.contains(target);
            }
            else if (rootValue.compareTo(target) < 0){ // if target is larger than the root
                BTree next = root.getRight();
                found = next.contains(target);
            }
        }
        return found;
    }

    /**
     * Traverse the tree, producing a list of all the values contained in the tree.
     * @using Preorder traversal.
     * @return a list of all the values in the tree.
     */
    @Override
    public List<T> traverse() {

        try {
            traversalList.add(root.value.toString());
        } catch (NullPointerException e) { }

        if (traversalList.size() < nodeCount){}

        BTree currentLeft = null;
        BTree currentRight = null;
        try {
```

```
            currentLeft = this.getLeft();
            currentRight = this.getRight();
        } catch (NullPointerException e) {

        }
        if (currentLeft != null){
            currentLeft.traverse();
        }
        if (currentRight != null){
            currentRight.traverse();
        }
        return traversalList;

    }
}
```

This is the test class made for the BinaryTree class.

```
package binaryTree;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import static org.junit.jupiter.api.Assertions.*;

class BinaryTreeTest extends BinaryTree {

    BinaryTree tree = new BinaryTree();
    ArrayList values = new ArrayList();
    ArrayList insertValues = new ArrayList();

    /**
     * Set up a new binary tree for testing.
     */
    @BeforeEach
    void setUp() {
        tree = new BinaryTree();
        values = new ArrayList();
        insertValues = new ArrayList();
        for (int i = 0; i < 50; i++) {
            if (i != 19){ // Don't add 19 - reserved for insert test.
                values.add(i);
            }
        }
        treeSetup(0, values.size());
    }

    /**
     * Create an unbalanced sorted binary tree for testing.
     * @param pointerLeft the leftmost value to insert.
     * @param pointerRight the rightmost value to insert
     */
    void treeSetup(int pointerLeft, int pointerRight) {
        double diff = pointerRight - pointerLeft;
        double midpoint = pointerLeft + Math.floor(diff / 2);
        if (diff > 0){
            Object newroot = values.get((int) midpoint);
            tree.insert((Comparable) newroot);
            insertValues.add(newroot);
            treeSetup(pointerLeft, (int) midpoint);
            treeSetup((int) (midpoint + 1), pointerRight);
        }
    }


    /** Test that 5 is within the tree
     * @return True if 5 is found.
```

```
     */
    @Test
        void contains5Test() {
        assertTrue (tree.contains(5));
    }

    /** Test that 0 is within the tree
     * @return True if 0 is found.
     */
    @Test
    void contains0Test() {
        assertTrue (tree.contains(0));
    }

    /** Test that 17 is within the tree
     * @return True if 17 is found.
     */
    @Test
    void contains17Test() {
        assertTrue (tree.contains(17));
    }

    /** Test that 50 is not within the tree
     * @return True if 50 is NOT found. (assertFalse = True if condition is false)
     */
    @Test
    void contains50Test() {
        assertFalse (tree.contains(50));
    }

    /**
     * test that 50 was correctly inserted into the tree.
     * @return True if 50 is found after being added.
     */
    @Test
    void insert50Test(){
        tree.insert(50);
        assertTrue(tree.contains(50));
    }

    /**
     * Test that 19 was correctly added to the tree.
     * @return True if 19 is found after being added.
     */
    @Test
    void insert19Test(){
        tree.insert(19);
        assertTrue(tree.contains(19));
    }

    /**
     * Test that the correct order is returned when the tree is traversed
     * No parameter is used. The traversal will always begin at the root node of the tree.
     * @return True if the traverse() matches the literal.
     */
    @Test
    void traverseTest() {
        List returned = tree.traverse();
        List<String> expected = Arrays.asList("25", "12", "6", "3", "1", "0", "2", "5", "4",
"9", "8", "7",
                "11", "10", "18", "15", "14", "13", "17", "16", "22", "21", "20", "24", "23",
"38", "32", "29",
                "27", "26", "28", "31", "30", "35", "34", "33", "37", "36", "44", "41", "40",
"39", "43", "42", "47", "46", "45", "49", "48");   // Literal expected output.

        assertEquals(expected , returned);

    }

}
```

I chose these tests as they test a wide range of the tree, including all left checks, all right checks, left and right checks and nodes not within the tree.

I tested the traverse() using a literal string as the traverse() method will always produce the same output from a tree as it will look left then right for each node. This expected result was calculated by hand using the algorithm written within the treeSetup() method and this is the tree that was made, and then manually traversed to create the expected result.

Self-Assessment

Rating - 5

I would give myself 5/5 for this weeks work as I have fully implemented and tested a working solution to the task presented. I believe that I have fully documented the code and tests with relevant and useful documentation which would allow another person to develop and maintain this code, although further development of the documentation is likely possible but not required in my opinion. The tests cover many possible use cases, although covering all is not possible. I believe I covered many cases required and that it proves the integrity of my solution.

Week 10

This is the code for the implementation of the Depth First Traversal.

```
package graph;

import java.util.*;

public class DepthFirstTraversal<T> extends AdjacencyGraph<T> implements Traversal<T> {

    ArrayList<T> toDoList = new ArrayList<>();
    ArrayList<T> visited = new ArrayList<>();

    /**
     * This is the main traverse method being used to get the traversal list.
     * @param root that is currently being traversed.
     * @return list of nodes visited in order of being visited.
     * @throws GraphError
     */
    synchronized public List<T> traverse(T root) throws GraphError {
        Set<T> nodes = getNodes();
        if (!nodes.contains(root)){
            throw new IndexOutOfBoundsException("Root was not within the Graph.");
        }
        if (!visited.contains(root)) {
            visited.add(root);
            Set<T> children = getNeighbours(root);
            for (T childNode : children) {
                if (!visited.contains(childNode)) {
                    toDoList.add(childNode);
                }
            }
        }
        try {
            for(T node : toDoList){
                toDoList.remove(node);
                traverse(node);
            }
        } catch (ConcurrentModificationException e) {}
        if (toDoList.isEmpty()){
            traverse();
        }
```

```
        return visited;
    }

    /**
     * This is a generic traverse() method called if no node is given.
     * @return list of nodes visited in order of being visited.
     * @throws GraphError
     */
    @Override
    public List<T> traverse() throws GraphError {
        T node = getUnvisited();
        if (node != null){traverse(node); }
        return visited;
    }

    /**
     * This method gets a node that hasn't been visited.
     * @return a random node from within the unvisited list
     */
    private T getUnvisited() {
        int nodeCount = getNoOfNodes();
        ArrayList<T> unvisited = new ArrayList<T>();
        if (nodeCount > visited.size()){
            ArrayList<T> listNodes = new ArrayList<T>();
            listNodes.addAll(getNodes());
            for(T node : listNodes) {
                if (!visited.contains(node)) {
                    unvisited.add(node);
                }
            }
        }
        else if (nodeCount == visited.size()){unvisited = null;}
        try {
            return   unvisited.get(new Random().nextInt(unvisited.size()));
        } catch (Exception e) {
            return null;
        }
    }
}
```

This is the code created for the test data.

```
package graph;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.util.HashSet;

import java.util.Set;

import static org.junit.jupiter.api.Assertions.*;

class DepthFirstTraversalTest <T> {

    DepthFirstTraversal graph = new DepthFirstTraversal();

    /**
     * Setup a graph before each of the tests.
     */
    @BeforeEach
    void setUp(){
        graph = new DepthFirstTraversal();
        try {
            graph.add(0);
            graph.add(1);
            graph.add(2);
            graph.add(3);
            graph.add(4);
        } catch (GraphError graphError) {
```

```
            System.out.println("FAILED!");
            graphError.printStackTrace();
        }

        // Add the links between the nodes.
        try {
            graph.add(0,1);
            graph.add(0,2);
            graph.add(0,3);
            graph.add(1,4);
            graph.add(3,2);
            graph.add(2,4);
        } catch (GraphError graphError) {
            System.out.println("FAILED!");
            graphError.printStackTrace();
        }
    }

    /**
     * Add 9 to an empty graph.
     * @throws GraphError
     */
    @Test
    void addTest() throws GraphError {
        graph = new DepthFirstTraversal();
        graph.add(9);
        Set expected = new HashSet();
        expected.add(9);
        assertEquals(expected, graph.getNodes());
    }

    /**
     * Traverse the created graph without an initial node.
     * @return True if all nodes are traversed.
     * @throws GraphError
     */
    @Test
    void traverseTest() throws GraphError {

        Set<T> results = new HashSet<T>(graph.traverse());
        Set<T> expected = graph.getNodes();
        boolean passed = true;
        for (T node : expected){
            if (!results.contains(node)){
                passed = false;
                break;
            }
        }
        assertTrue(passed);
    }

    /**
     * Traverse the created graph with an initial node of 1.
     * @return True if all nodes are traversed.
     * @throws GraphError
     */
    @Test
    void traverse1Test() throws GraphError {
        Set<T> results = new HashSet<T>(graph.traverse(1));
        Set<T> expected = graph.getNodes();
        boolean passed = true;
        for (T node : expected){
            if (!results.contains(node)){
                passed = false;
                break;
            }
        }
        assertTrue(passed);
    }

    /**
     * Traverse the created graph with an initial node not in the graph (10).
```

```
     * @return True if an IndexOutOfBoundsException is thrown.
     * @throws IndexOutOfBoundsException
     */
    @Test
    void traverse10Test() {
        assertThrows(IndexOutOfBoundsException.class, ()->{
            graph.traverse(10);
        });
    }
}
```

Self-Assessment

Rating - 5

I would give myself 5/5 for this weeks work as I have fully tested the solution I created. I tested that an out of bounds node throws an IndexOutOfBoundsException as this is a more specific exception than the given GraphException as it is not an exception caused by the graph but by the index (node) given by the user. The rest of the tests confirm that the correct traversal is shown if the user traverses from any existing node. I also tested that values can be added to the graph as I was initially having issues with this (I left them in to confirm that it was still working later on).

I believe that my implementation follows good programming practice as I have separated my methods into blocks that achieve a single task, making each method atomic, and created a traversal() method for when a node is given and for when no node is specified. This means that future bug fixes are easier as the code is separated into sections and has clear documentation for each.

Week 11

This is the code for the implementation of the Reference count Topological sort.

```
package graph;

import java.util.*;

public class ReferenceCountTopologicalSort <T> extends AdjacencyGraph implements TopologicalSort
{

    private HashMap <T, Integer> referenceCounts = null;
    private ArrayList<T> visited = new ArrayList<>();

    /**
     * Creates a referenceCounts HashMap to store the number of times a
     * node is connected to from another.
     * Requires a graph to be created so that it can generate
     * @throws GraphError
     */
    private void setupReferenceCounts() throws GraphError {
        referenceCounts = new HashMap<>();
        ArrayList<T> nodes = new ArrayList<>();
        nodes.addAll(this.getNodes());
        if (nodes.isEmpty()){throw new GraphError("There were no nodes in the graph");}
        for (T node : nodes){
            referenceCounts.put(node, 0);
        }

        for (int i = 0; i < nodes.size(); i++){
            ArrayList<T> neighbours = new ArrayList<>();
            neighbours.addAll(getNeighbours(nodes.get(i)));
            for (T neighbour : neighbours){
                Integer value = referenceCounts.get(neighbour) + 1;
                referenceCounts.replace(neighbour, value);
```

```
            }
        }
    }

    /**
     * Find any node with 0 connections that isn't within the visited ArrayList.
     * If no nodes conform then the graph is cyclic.
     * @return T Node
     * @throws GraphError if the graph is Cyclic.
     */
    private T getZeroReferenceCount() throws GraphError {
        T nextNode = null;
        ArrayList<T> nodes = new ArrayList<>();
        nodes.addAll(this.getNodes());
        for (int i = 0; i < referenceCounts.size(); i++){
            if (referenceCounts.get(nodes.get(i)) == 0 && !visited.contains(nodes.get(i))){
                nextNode = nodes.get(i);
                break;
            }
        }
        if (nextNode == null){
            throw new GraphError("The graph was Cyclic! ");
        }
        return nextNode;
    }

    /**
     * Creates a ReferenceCountTopologicalSort of an Acyclic graph.
     * @param 'an Acyclic graph'
     * @return A Topological sort of a graph
     * @throws GraphError
     */
    @Override
    public List getSort() throws GraphError {
        setupReferenceCounts();
        ArrayList <T> sorted = new ArrayList<>();
        while (sorted.size() < getNoOfNodes()) {
            T currentNode = getZeroReferenceCount();
            sorted.add(currentNode);
            UpdateReferenceCounts(currentNode);

        }
        return sorted;
    }

    /**
     * Updates the reference count to store the number of
     * unprocessed references to a node.
     * Removes the references from the currentNode.
     * Remove the old data and store the new value as there is no Update() method.
     * @param currentNode as the current node as the target.
     * @throws GraphError
     */
    private void UpdateReferenceCounts(T currentNode) throws GraphError {
        ArrayList<T> neighbours = new ArrayList<>();
        neighbours.addAll(getNeighbours(currentNode));
        visited.add(currentNode);
        for (T neighbor : neighbours){
            Integer value = referenceCounts.get(neighbor);
            referenceCounts.remove(neighbor);
            referenceCounts.put(neighbor, value-1);
        }
    }
}
```

This is the test code for this weeks exercises

```
package graph;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.util.*;

import static org.junit.jupiter.api.Assertions.*;

class ReferenceCountTopologicalSortTest {

    ReferenceCountTopologicalSort<Integer> graph = new ReferenceCountTopologicalSort<>();

    /**
     * Setup an acyclic graph of size 5.
     * The graph uses Integers as nodes.
     * @throws GraphError
     */
    @BeforeEach
    void setUp() throws GraphError {

        graph = new ReferenceCountTopologicalSort<>();

       // Add the nodes to the graph
        graph.add(0);
        graph.add(1);
        graph.add(2);
        graph.add(3);
        graph.add(4);

        // Add the links between the nodes.
        graph.add(0,1);
        graph.add(0,2);
        graph.add(0,3);
        graph.add(1,4);
        graph.add(3,2);
        graph.add(2,4);

    }

    /**
     * Test that the graph was successfully created.
     * @return True if the graph was created.
     */
    @Test
    void setupTest(){

        Hashtable<Integer, Set<Integer>> expected = new Hashtable<>();

        for (Integer i = 0; i < 5; i ++){ expected.put(i, new HashSet<>()); }
        expected.get(0).add(1);
        expected.get(0).add(2);
        expected.get(0).add(3);
        expected.get(1).add(4);
        expected.get(3).add(2);
        expected.get(2).add(4);

        assertEquals(expected, graph.getGraph());
    }

    /**
     * Test that getSort() contains all correct nodes.
     * @throws GraphError
     */
    @Test
    void valueTest() throws GraphError {
        List<Integer> sort = graph.getSort();
        List<Integer> expected = new ArrayList<>();
        for (int i = 0; i < 5; i++){expected.add(i);}
        boolean success = true;
```

```
        for (Integer node : expected){
            if (!sort.contains(node)){
                success = false;
                break;
            }
        }
        assertTrue(success);
    }

    /**
     * Test that a GraphError is thrown for a cyclic graph.
     * @return True if the GraphError is thrown.
     * @throws GraphError due to cyclic graph.
     */
    @Test
    void cycleGraphTest() throws GraphError {
        graph.add(5);
        graph.add(0,5);
        graph.add(5,0);

        assertThrows(GraphError.class,()-> graph.getSort());
    }

    /**
     * Graph found at:
     * https://cs.stackexchange.com/questions/2186/dependency-graph-acyclic-graph
     * This link was used to find a large graph with Chars as nodes.
     *
     * Test that nodes can be of any type, and that larger
     * graphs than the previous test will work.
     * @return True if the nodes are all returned.
     * @throws GraphError thrown if the graph is cyclic.
     */
    @Test
    void largeDataTest() throws GraphError {
        graph = new ReferenceCountTopologicalSort<>();
        List<Character> expected = new ArrayList<>();
        for (char alphabet = 'A'; alphabet < 'N'; alphabet++){
            graph.add(alphabet);
            expected.add(alphabet);
        }

        graph.add('A', 'C');
        graph.add('A', 'D');
        graph.add('C', 'F');
        graph.add('C', 'G');
        graph.add('C', 'H');
        graph.add('G', 'K');
        graph.add('H', 'K');
        graph.add('D', 'I');
        graph.add('B', 'E');
        graph.add('E', 'H');
        graph.add('E', 'I');
        graph.add('E', 'J');
        graph.add('I', 'L');
        graph.add('I', 'M');
        graph.add('J', 'M');

        List<Character> sort = graph.getSort();
        boolean success = true;
        for (char node : expected){
            if (!sort.contains(node)){
                success = false;
                break;
            }
        }
        assertTrue(success);
    }

    /**
     * Test the outcome of a mixed node graph (Integers and Chars).
     * This uses the same graph as the SetUp() but Even numbers are replaced with Chars.
```

```
     * Use of Object type is used in testing to check that a list can be
     * returned with multiple types.
     * @return True if the nodes are successfully sorted.
     * @throws GraphError if the graph is cyclic.
     */
    @Test
    void mixedNodeTest() throws GraphError {
        graph = new ReferenceCountTopologicalSort<>();
        List<Object> expected = new ArrayList<>();

        graph.add('a');
        expected.add('a');
        graph.add(1);
        expected.add(1);
        graph.add('b');
        expected.add('b');
        graph.add(3);
        expected.add(3);
        graph.add('c');
        expected.add('c');

        // Add the links between the nodes.
        graph.add('a',1);
        graph.add('a','b');
        graph.add('a',3);
        graph.add(1,'c');
        graph.add(3,'b');
        graph.add('b','c');

        List<Object> sort = graph.getSort();
        boolean success = true;
        for (Object node : expected){
            if (!sort.contains(node)){
                success = false;
                break;
            }
        }
       assertTrue(success);

    }

    /**
     * Test the outcome of an empty grpah.
     * @return True if the Exception is thrown
     * @throws GraphError as there are no nodes in the graph.
     */
    @Test
    void emptyGraphTest() throws GraphError {
        graph = new ReferenceCountTopologicalSort<>();
        assertThrows(GraphError.class,()-> graph.getSort());
    }

    /**
     * Test the outcome of a search if a graph has 1 node and no edges.
     * @return True if the node is returned.
     * @throws GraphError if the graph is cyclic.
     */
    @Test
    void singleNodeTest() throws GraphError {
        graph = new ReferenceCountTopologicalSort<>();
        List<Integer> expected = new ArrayList<>();
        expected.add(0);
        graph.add(0);
        List<Integer> sort = graph.getSort();
        assertEquals(expected, sort);
    }

    /**
     * Test that a GraphError is thrown if a graph has 1 node and 1 edge.
     * @return True if the exception is thrown.
     * @throws GraphError as the graph is Cyclic.
     */
```

```
    @Test
    void singleNodeCycleTest() throws GraphError {
        graph = new ReferenceCountTopologicalSort<>();
        graph.add(0);
        graph.add(0,0);
        assertThrows(GraphError.class,()-> graph.getSort());
    }
}
```

Self-Assessment

Rating - 5

I would give myself 5/5 for this weeks work as I have fully implemented and tested the solution using scenarios that are common, uncommon, and result in exceptions being thrown. Common use-cases of my solution would be tests that require single data types for the nodes, be it Integers or Chars. These result in a working graph that can be successfully sorted. An uncommon use-case of my solution is the test of both Char and Integer nodes. This resulted in a successful test as it resulted in a sort of [a, 1, 3, b, c], the same as the original graph (where a, b, and c replaced 0, 2, and 4 respectively). As these are the same the processing is the same – the only change was the node identifier (the edges remained the same).

I created some tests that throw exceptions, including empty graphs and cyclic graphs. These are included as these may be attempted within a common use-case as an accidental cycle or a malicious attempt to break the system.

I tested a graph with 1 node and no edges and a graph with 1 node and 1 edge. This was to test that the exception was thrown by the cycle and not the node. This worked as expected so I can deduce that a cyclic graph will always be rejected by the program, regardless of how many nodes the graph has (1 or more) and that the solution will work with any number of nodes (1 or more).