Assignment 1 – Documentation

Contents

Task 1:

The first task was to implement the Word Class by creating multiple methods (noting that every word has to be lower case and spaces removed) and testing their success rate.

Planned UML Diagram (based off task sheet):

+ Word
- String word - String wordToNumber
+ Word(String word) + String setWord(String word) + String getWord() + char getDigit(char letter) + void setWordToNumber(String word) + String toString()

Final UML Diagram:

+ Word
- String word - String wordToNumber - Hashmap<Character, Integer> valueMap
+ Word(String word) + createMap() + addTelephoneKeys(String characters, int value) + String setWord(String word) + String getWord() - boolean checkInput(String word) + char getDigit(char letter) + void setWordToNumber(String word) + String toString() + boolean equals(Object o) - class InputException extends Exception

The final UML Diagram demonstrates the changes that were made throughout the implementation of the first task.

The methods (and their purposes) added include;

- createMap – creates the hash map that makes sorting more efficient for searching
- addTelephoneKeys – adds all the elements to the hash map
- checkInput – ensures only alphanumeric characters are entered for the word
- equals – overwrites the equals method for the Word class
- InputException (class) – handles any input exceptions throughout the class

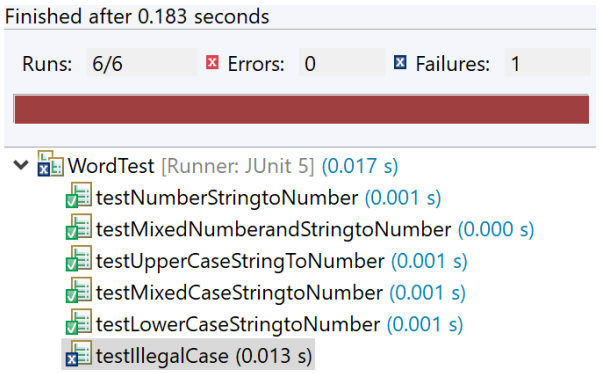
These methods were added to ensure the code written runs efficiently and fulfils the task requirements.

Test Cases:

For the Word class, I tested using JUnit testing (WordTest.java). I tested multiple cases including an Upper Case string, Lower Case string, Mixed Case string, Number string, Mixed Numbed and Letter string, and one case that should throw an exception. The test can be seen below:

```
public class WordTest extends TestCase {
	@Test public void testUpperCaseStringToNumber() {
		Word testWord = new Word("PIZZAGO");
		assertEquals("7499246", testWord.getWordToNumber());
	}

	@Test public void testLowerCaseStringtoNumber() {
		Word testWord = new Word("pizzago");
		assertEquals("7499246", testWord.getWordToNumber());
	}

	@Test public void testMixedCaseStringtoNumber() {
		Word testWord = new Word("PizZagO");
		assertEquals("7499246", testWord.getWordToNumber());
	}

	@Test public void testNumberStringtoNumber() {
		Word testWord = new Word("1234567890");
		assertEquals("1234567890", testWord.getWordToNumber());
	}

	@Test public void testMixedNumberandStringtoNumber() {
		Word testWord = new Word("PizZA2gO");
		assertEquals("74992246", testWord.getWordToNumber());
	}

	@Test public void testIllegalCase() throws Exception {
		Word testWord = new Word("(PizZA2gO!");
		//should fail > does fail
		assertEquals("Invalid input",
		testWord.setWordToNumber(testWord.getWord()));
	}
}
```

Results:



The results were as expected and each test case passed, returning the single failure that is thrown by the exception. This proves that the Word class is working for all legal cases, throwing exceptions for illegal cases and fulfilling the Task 1 requirements.

Initially, for Task 1, a hashMap was not used for this task with multiple if statements used (see code below). However, upon reflection it was confirmed that this is not the most efficient method of changing a letter/number into a digit and the getDigit method was changed and the above methods implemented to improve efficiency across the program.

```
public char getDigit(char letter) {
            int codePoint = (int) letter;
            char digit;

            if (codePoint >=97 && codePoint <=99)
                  digit = '2';
            else if (codePoint >= 100 && codePoint <=102)
                  digit = '3';
            else if (codePoint >=103 && codePoint <=105)
                  digit = '4';
            else if (codePoint >=106 && codePoint <=108)
                  digit = '5';
            else if (codePoint >=109 && codePoint <=111)
                  digit = '6';
            else if (codePoint >=112 && codePoint <=115)
                  digit = '7';
            else if (codePoint >=116 && codePoint <=118)
                  digit = '8';
            else if (codePoint >=119 && codePoint <=122)
                  digit = '9';
            else if (codePoint == 48)
                  digit = '0';
            else if (codePoint == 49)
                  digit = '1';
            else if (codePoint == 50)
                  digit = '2';
            else if (codePoint == 51)
                  digit = '3';
            else if (codePoint == 52)
                  digit = '4';
            else if (codePoint == 53)
                  digit = '5';
            else if (codePoint == 54)
                  digit = '6';
            else if (codePoint == 55)
                  digit = '7';
            else if (codePoint == 56)
                  digit = '8';
            else if (codePoint == 57)
                  digit = '9';
            else {
                  digit = ' ';

            return digit;
      }
```

Task 2:

Task 2 was to update the PhoneWords class and implement the addWord and isKnown methods. This task was designed to add words to the collection, by checking whether or not they exist (isKnown method). The getNumWords is also implemented, returning the number of words in the collection (dictList).

UML Diagram(s) before and after the implementation of Task 2:

```
+ PhoneWords
- boolean numSorted
- boolean numUniqueSorted
+ PhoneWords(boolean isWordsFileProvided)
- static void loadDict(String name, PhoneWords pw)
```

```
+ PhoneWords
- boolean numSorted
- boolean numUniqueSorted
- List<Word> dictList = new ArrayList<Word>()
+ PhoneWords(boolean isWordsFileProvided)
- static void loadDict(String name, PhoneWords pw)
+ void addWord(String w)
+ boolean isKnown(Word word)
+ int getNumWords()
```

Testing for the isKnown method:

```
//Testing for isKnown
	Word w = new Word("CAB");
	PhoneWords pw = new PhoneWords(true);
	System.out.println(pw.isKnown(w));
	Word w1 = new Word("222");
	System.out.println(pw.isKnown(w1));
```

ID	Description	Input	Expected Output	Actual Output	Pass/Fail
1	Lowercase string	“cab”	True	True	Pass
2	Uppercase string	“CAB”	True	False	Fail – have not allowed for uppercase letters in my search
3	Uppercase string	“CAB”	True	True	Pass – altered the isKnown code to include a toLowerCase() to ensure the string can be found
4	Illegal case (throw an exception)	“bab90;”	Invalid Output	Invalid Output False	Pass – the code threw the exception and returned false for the isKnown boolean for an illegal input
5	Case to return false	“222”	False	False	Pass – returns false because the word is never changed into its number equivalent (as the method states)

Testing the addWord method:

```
//Testing for addWord
	System.out.println("Testing for addWord");
	PhoneWords pw = new PhoneWords(true);
	pw.addWord("popltip");
	System.out.println(pw.numWords("popltip"));
	pw.addWord("babble");
	pw.addWord("CABLELB");
	System.out.println(pw.numWords("CABLELB"));
	pw.addWord("BABBLE");
	pw.addWord("po90%");
```

ID	Description	Input	Expected Output	Actual Output	Pass/Fail
1	Lowercase string that does not exist	“popltp”	Word added 1	word added successfully 1	Pass Note: the numWords case was altered for testing (getWordToNumber changed to getWord)
2	Lowercase string that does exist	“babble”	Word is known	word is known	Pass
3	Uppercase string that does not exist	“CABLEB”	Word added 1	word added successfully 1	Pass (Same note as ID 1)
4	Uppercase string that does exist	“BABBLE”	Word is known	word is known	Pass
5	Illegal case	“po90%”	Invalid Input (Throw exception)	Invalid Input	Pass

Testing the getNumWords method:

```
//Testing for get numWords
	PhoneWords pw = new PhoneWords(true);
	System.out.println(pw.getNumWords());
	pw.addWord("cababababab");
	System.out.println(pw.getNumWords());
```

The getNumWords method was tested by first retrieving the number of words in the dictList, before adding one word and retrieving the number of words again.

Results:

```
Number of words prior to adding a word: 39629
Number of words after adding a word: 39630
```

The results prove that the getNumWords method is working as a single word has been successfully added.

Task 3:

Task 3 involved implementing listWords to return a list of words matching the number provided and numWords to return the number of words matching the number provided. This was achieved through the implementation of extra methods of isValidNumber and countOccurance.

- isValidNumber - determines whether or not the provided number has valid usage of brackets and + (for the area code) or has random symbols
- countOccurance – counts the occurrences of a specific character within a given string (to search for duplicates of characters e.g. "+")

Updated UML Diagram:

+ PhoneWords
- boolean numSorted - boolean numUniqueSorted - List<Word> dictList = new ArrayList<Word>()
+ PhoneWords(boolean isWordsFileProvided) - static void loadDict(String name, PhoneWords pw) + void addWord(String w) + boolean isKnown(Word word) + int getNumWords() - static boolean isValidNumber(String num) - static int countOccurrence(String str, char c) + ArrayList<Object> listWords(String num) + int numWords(String string)

See testing on following page for added methods.

Testing the isValidNumber method: (static modifier was removed for testing)

```
//Testing the isValid method
	PhoneWords pw = new PhoneWords(true);
	System.out.println(pw.isValidNumber("(02) 6268 8000"));
	System.out.println(pw.isValidNumber("(02 6268 8000"));
	System.out.println(pw.isValidNumber("+61 6268 8000"));
	System.out.println(pw.isValidNumber("61 +2 6268 8000"));
	System.out.println(pw.isValidNumber("6268 8000"));
	System.out.println(pw.listWords("(22) 23"));
	System.out.println(pw.isValidNumber("6268 *90!"));
```

ID	Description	Input	Expected Output	Actual Output	Pass/Fail
1	String that should work with brackets	(02) 6268 8000	true	true	Pass
2	String that contains illegal use of brackets	(02 6268 8000	false	false	Pass
3	String that should work with + symbol	+61 6268 800	true	true	Pass
4	String that contains illegal use of +	61 + 2 6268 890	false	false	Pass
5	String that works with only numbers	6268 8000	true	true	Pass
6	String that contains illegal symbols	6268 *90!	false	true	Fail – did not account for extra symbols other than brackets and +
7	String that contains illegal symbols	6268 *90!	False	false	Pass – new for loop added to test for illegal symbols

Testing the countOccurrence method: (static modifier was removed for testing)

```
//Testing the countOccurrence method
	PhoneWords pw = new PhoneWords(true);
	System.out.println(pw.countOccurrence("(02) 2628 8000", ')'));
	System.out.println(pw.countOccurrence("+61 2+28 8000", '+'));
	System.out.println(pw.countOccurrence("(02) 2628 8000", '+'));
```

ID	Description	Input	Expected Output	Actual Output	Pass/Fail
1	String that should return 1 count occurrences	“(02) 2628 8000”, ‘)’	1	1	Pass
2	String that should return 2 count occurrences	“+61 2+28 8000”, ‘+’	2	2	Pass
3	String that should return 0 count occurrences	“(02) 2628 8000”, ‘+’	0	0	Pass

Testing the listWords method:

```
//Testing the listWords method
	PhoneWords pw = new PhoneWords(true);
	System.out.println(pw.listWords("2223"));
	System.out.println(pw.listWords("6"));
	System.out.println(pw.listWords("111"));
	pw.listWords("09!45"));
```

ID	Description	Input	Expected Output	Actual Output	Pass/Fail
1	String that should find two words	“2223”	Two words – babe and abbe	`[Word [word=babe, wordToNumber=2223], Word [word=abbe, wordToNumber=2223]]`	Pass
2	String that should find two words (that are letters)	“6”	Two words – n and o	`[Word [word=n, wordToNumber=6], Word [word=o, wordToNumber=6]]`	Pass
3	String that should find no result	“111”	An empty array	`[]`	Pass
4	String with brackets	“(22) 23”	Two words – babe and abbe	`[Word [word=babe, wordToNumber=2223], Word [word=abbe, wordToNumber=2223]]`	Pass
5	Illegal case	“09!45”	Throw Exception	Exception thrown	Pass

Testing the numWords method:

```
	//Testing the numWords method
	PhoneWords pw = new PhoneWords(true);
	 System.out.println(pw.numWords("2223"));
	 System.out.println(pw.numWords("6"));
	 System.out.println(pw.numWords("111"));
	 System.out.println(pw.numWords("09!45"));
```

ID	Description	Input	Expected Output	Actual Output	Pass/Fail
1	Method should return int 2	“2223”	2	2	Pass
2	Method should return int 2	“6”	2	2	Pass
3	Method should return int 0	“111”	0	0	Pass
4	Method should return int 0	“09!45”	0	0	Pass

Task 4:

Final PhoneWords UML Diagram

+ PhoneWords
- boolean numSorted - boolean numUniqueSorted - List<Word> dictList = new ArrayList<Word>()
+ PhoneWords(boolean isWordsFileProvided) - static void loadDict(String name, PhoneWords pw) + void addWord(String w) + boolean isKnown(Word word) + int getNumWords() - static boolean isValidNumber(String num) - static int countOccurrence(String str, char c) + ArrayList<Object> listWords(String num) + int numWords(String string) + boolean setNumSorted(boolean sorted) + boolean getNumSorted() + boolean setNumUniqueSorted(boolean uniqueSorted) + boolean getNumUniqueSorted() - class InputException extends Exception

Note: listWords was altered in order to effectively test these methods (Else and print statements added). See changes below:

```
if (numSorted && numUniqueSorted) {
	System.out.println("Running listWords");
	if(!isValidNumber(num))
		throw new InputException();
	for(int i = 0; i < getNumWords(); i++)
		if(num.equals(dictList.get(i).getWordToNumber())) {
			ar.add(dictList.get(i));}
}
else {
	System.out.println("numSorted: " + getNumSorted() + " " +
	"\nnumUniqueSorted: " + getNumUniqueSorted());
}
```

Testing for numSorted and numUniqueSorted:

```
//Testing the numSorted and numUniqueSorted methods
PhoneWords pw = new PhoneWords(true);
pw.setNumSorted(true);
pw.setNumUniqueSorted(true);
pw.listWords("2223");
pw.setNumSorted(false);
pw.setNumUniqueSorted(true);
pw.listWords("2223");
pw.setNumSorted(true);
pw.setNumUniqueSorted(false);
pw.listWords("2223");
pw.setNumSorted(false);
pw.setNumUniqueSorted(false);
pw.listWords("2223");
```

ID	Description	Input	Expected Output	Actual Output	Pass/Fail
1	Both numSorted and numUniqueSorted are true	Set both variables to true	“Running listWords”	`Running listWords`	Pass
2	numSorted is true, numUniqueSorted is false	Set numSorted to true, numUniqueSorted to false	“numSorted: true numUniqueSorted: false”	`numSorted: true umUniqueSorted: false`	Pass
3	numSorted is false, numUniqueSorted is true	Set numSorted to false, numUnqiueSorted to true	“numSorted: false numUniqueSorted: true”	`numSorted: false numUniqueSorted : true`	Pass
4	Both numSorted and numUniqueSorted are false	Set both variables to false	“numSorted: false numUniqueSorted: false”	`numSorted: false numUniqueSorted : false`	Pass

Extra working (Task 4):

For task 4 – my initial thought was that the Phonewords class had to be altered so that numSorted and numUniqueSorted booleans were updated by the code once a number was inputed. This meant that if the user entered a number that was not sorted, it would be sorted and then if the user wanted no duplicates, setNumUniqueSorted would also remove all duplicates. However, after talking to Saber, it was discovered that this was not required (but I had already implemented it). Below is the code to implement the above idea of both sorting and uniquely sorting a number for the user.

```
/**
 * sorts the string of numbers into ascending order
 * @param num - the string that will be sorted
 * @return returns a sorted string
 */
public String isSorted(String num) {
   char temp;
   String isSorted = "";
   char[] numArray = num.toCharArray();

   for (int i = 0; i < numArray.length; i++)
    {
        for (int j = i + 1; j < numArray.length; j++)
        {
            if (numArray[i] > numArray[j])
            {
                temp = numArray[i];
                numArray[i] = numArray[j];
                numArray[j] = temp;
            }
        }
    }
```

```
      //rearranges the array so it prints a continuous string i.e. without
brackets or commas
       String printArray = new String(numArray);
       isSorted = String.join("", printArray);
       return isSorted;
    }

    /**
     * calls the method that removes all redundant numbers from the String num
(after sorting)
     * @param num - the string that has duplicate numbers
     * @return return a new string without duplicate numbers
     */
    public String isUniqueSorted(String num) {
       num = isSorted(num);
       return removeDuplicates(num);
    }

    /**
     * removes all redundant numbers from the sorted string
     * @param str - the string that has duplicate numbers
     * @return returns a string without duplicates
     */
    public String removeDuplicates(String str) {
       char[] chars = str.toCharArray();
       Set<Character> charSet = new LinkedHashSet<Character>();
       for (char c : chars) {
           charSet.add(c);
       }

       String sb = "";;
       for (Character character : charSet) {
           sb += character;
       }
       return sb;
    }

    /**
     * finds if the string is sorted
     * @param str - the string that may or may not be sorted
     * @return returns a true or false dependent on if the string is sorted
     */
    public boolean findifSorted(String str) {
       boolean sorted = true;
       int num = Integer.parseInt(str);
       int currentDigit = num % 10;
        num = num/10;

        while(num>0){
            if(currentDigit <= num % 10){
                sorted = false;
                break;
            }
            currentDigit = num % 10;
            num = num/10;
        }

       return sorted;
```

```
    }

    /**
     * finds if there are duplicates in the string
     * @param str - string to be tested for duplicates
     * @return returns true or false dependent on if there are duplicates
     */
    public boolean findifDuplicates(String str) {
       boolean duplicatesfound = true;

       for (int i = 0; i < str.length(); i++)
            for (int j = i + 1; j < str.length(); j++)
                if (str.charAt(i) == str.charAt(j))
                    return false;

       return duplicatesfound;
    }
```

To allow this code to work the following would also have to be implemented (changed setNumSorted and setNumUniqueSorted):

```
    /**
     * sets the boolean for whether or not a string is sorted
     * @param num - the string to be tested to see if it is sorted
     * @return returns true or false dependent on if the string is sorted
     */
    public boolean setNumSorted(String num) {
       return numSorted = findifSorted(num);
    }

    /**
     * sets the boolean for whether or not there are redundant numbers in the
string
     * @param num - the string to be tested for redundant numbers
     * @return returns true or false dependent on if the string has redundant
numbers
     */
    public boolean setNumUniqueSorted(String num) {
       return numUniqueSorted = findifDuplicates(num);
    }
```

The above code required me to search for effective and efficient ways to sort through a string. I finally decided on a bubble sort method for the isSorted method (possibly not for its efficiency – but for its simplicity and ease of application) and then removeDuplicates method utilises a LinkedHashSet (as it does not allow duplicates) and then rebuilds the string from that set.

The findifSorted and findifDuplicates methods were created for ease of use for the program – as if the number is already sorted the isSorted method does not need to be called and a similar approach applies to the isUniqueSorted method (if there are no duplicates, there is no requirement to run the method).

Task 5:

The purpose of task 5 was to improve the PhoneWords class by finding the most time efficient way of adding all the words in words.txt to a collection.

My choice options for collections included:

- List - ArrayList
- Stack
- Queue - LinkedList
- Deque - LinkedList
- Set – HashSet

In order to efficiently test the time taken to fill each type of Collection a TestCollectionTime class was created. Within this class, the choice of collection was set to a string and then the dictionary added to the different type of collections depending on the collection chosen by the string.

The below results show that a Set is the fastest collection type for filling the collection.

```
Time for List collection:32082
Time for Stack collection:38003
Time for Queue collection:100436
Time for Deque collection:111962
Time for Set collection:581
```

However, we must be aware that the collection will also be sorted in the NewPhoneWords class.

Through minor testing it was discovered that it was difficult to implement a Set within the NewPhoneWords class. There were issues returning values within the listWords method and, after research, it was discovered the set would have to be locally turned into an array in order to search and then converted back to an array. If the task was only to add to a collection then a Set would be the most obvious choice. However, the collection must be implemented across the program and as such a Set is not economical. Instead, an ArrayList was chosen as the most economical way of completing this assignment, in terms of adding all the words and searching to find a word.

ArrayList was chosen as the most efficient collection because;

- It is the second fastest to the Set collection and issues do not arise when searching for a string within the Array (as was found with the Set)
- An ArrayList is extremely fast to access with O(1) performance > meaning that no matter how much data is entered, it will execute in constant time
- The data is stored in such a way that is access in a certain order and elements can be access are replaced by index

Implementation of a Set:

```
    //creates a new HashSet
    Set<Word> hashWords = new HashSet<Word>();
```

The code below shows the complexity and how it was necessary to create a list when trying to search for a single word.

```
//creates a new array list called  newList
    ArrayList<Object> newList = new ArrayList<Object>();
    Iterator<Word> middle = hashWords.iterator();

    //find the same word as getWordToNumber > add to newList
    while(middle.hasNext())
    {
        Word temp = middle.next();

        if (num.equals(temp.getWordToNumber()))
        {
            newList.add(temp);
        }
    }
    return newList;
```

The methods of numWordsTimeTestHash and numWordsTimeTestList (see NewPhoneWords for code) were implemented to compare the times taken to retrieve a word between the two collections; Set and List:

Main method code used for testing:

```
        //Testing the HashWord collection
        NewPhoneWords npw = new NewPhoneWords(true);
        System.out.println("HASH");
        npw.createHashSet();
        System.out.println(npw.numWordsTimeTestHash("2223"));
        System.out.println("LIST");
        System.out.println(npw.numWordsTimeTestList("2223"));
```

Result from testing:

```
HASH
2
22
LIST
2
14
```

The above result shows that it is quicker to run the numWords with a list (14 milliseconds) than with a set (22 milliseconds). This logic can also be applied to the ListWords class, as it requires a conversion to an array anyway and the isKnown class (where searching for a word is also required and will therefore take longer than an ArrayList).

I believe my initial implementation of the PhoneWords class used the best Collection and therefore, although NewPhoneWords was implemented with a Set – after thorough testing it does not utilise the most efficient collection.

Task 6:

I think it is very useful to design not only tests, but also general outlines (UML Diagrams) of code before implementing. By designing tests, it often easier to understand what the question is asking you to code and what exceptions must be thrown etc. Continuous testing is key to ensure that an error is not carried throughout the implementation of the project and can be fixed before it causes further errors.

Designing the tests sometimes led me to redesign my code, but often it was during the test process that bugs were discovered, as the tests detected these errors and they were subsequently corrected.

In the first task, the implementation of JUnit testing was very useful, as I was able to design all my tests prior and then check periodically throughout my implementation if certain tests were passing or which ones were still failing (i.e. which cases had I still not accounted for).