COMP0004 Coursework 1

Purpose: Writing a Java program with several classes to confirm that you understand the basics of object-oriented programming with Java.

Submission Deadline: 4pm 8th February 2021, upload a single zip file to Moodle. The zip file should contain the files from your IDEA project.

This is not required for the coursework but do this if you can: While working on your code, the project files and source code (no compiled code) should be stored in a GitHub repository, so that you are using version control properly. You should be making regular commits to the repository.

Marking: This part of the coursework is worth 20% of the module mark.

Inadequate (0-39)	Failed to demonstrate a basic understanding of the concepts used in the coursework, or answer the questions. There are fundamental errors, code will not compile, or nothing of significance has been achieved. (F)
Just Adequate (40-49)	Shows a basic understanding, sufficient to achieve a basic pass, but still has serious shortcomings. Code may compile but doesn't work properly. Not all concepts have been understood or all the core questions answered correctly. (D)
Competent (50-59)	Reasonable understanding but with some deficiencies. The code compiles and runs, the concepts are largely understood. This is the default for a straightforward, satisfactory answer. (C)
Good (60-69)	A good understanding, with possibly a few minor issues, but otherwise more than satisfactory. The code compiles, runs and demonstrates good design practice. Most expectations have been met. (B)
Very Good (70-79)	A very good understanding above the standard expectations, demonstrating a clear proficiency in programming and design. (A)
Excellent (80-100)	Programming at a level well above expectations, demonstrating expertise in all aspects. This level is used sparingly only where it is fully justified. (A+, A++)

Q1. Write a class StringArray that can store a collection of String object references (i.e., a collection of Strings). The data structure used in the class to store the Strings *must* be a Java array, you cannot use any other data structure like an ArrayList. The array must be private giving strong encapsulation.

From the external view seen by code using a StringArray, values (i.e., String references) in the StringArray should be stored sequentially and accessed via their index position like an array. There should be no gaps in the sequence and no unused index positions. It should be possible to store null references as values if a null is added to the StringArray or assigned to an index position.

The internal Java array does not have to be the exact size needed to store the references currently stored in the StringArray and can have unused space. It is preferable to have a larger array with some unused array elements, so that adding further references does not require a larger array to be created every time a new reference is added. The default size of the internal Java array holding the values might be 100, with the array size increased as necessary when more references are added. Think about the best strategy for increasing the array size. Should the size be doubled? Or increased by 100? Or by 10%. There are lots of possibilities, so choose one.

Provide the following methods (use exactly the names, parameter lists, and return types shown here):

- A constructor that creates an empty StringArray.

```
public StringArray()
```

- A constructor that creates a new StringArray containing a copy of the String object references stored in the parameter array.

```
public StringArray(StringArray a)
```

- Return the number of String references held in the StringArray, zero if empty. Note the size is determined by the stored references (number of Strings in the StringArray), not by the size of the internal array.

```
public int size()
```

- Return true if there are no Strings references stored in the StringArray, otherwise false.

```
public boolean isEmpty()
```

- Get the String reference at the given index position or return null if the index is invalid.

```
public String get(int index)
```

- Store a String reference at the given index position. The index must be for an element that exists in the StringArray, so that the previously stored String reference is overwritten. If the index is invalid nothing is changed.

```
public void set(int index, String s)
```

- Add a String reference to the end of a StringArray, immediately following the last element. The StringArray size increases by one. If the internal Java array is full, a new larger array is created, and the exiting values copied into it before the new String reference is added.

```
public void add(String s)
```

- Insert a String reference at the given index position by moving all elements after the index position forward by one to make space for the new value. The StringArray increases its size by one. If the index value is invalid nothing is changed. If the StringArray is empty a String reference can be inserted at index position zero. If the internal Java array is full, a new larger array is created, and the existing String references copied into it before the new reference is inserted.

```
public void insert(int index, String s)
```

- Remove a String reference at the given index position by moving all elements after the index position back by one to close up the gap left by removing the reference. The StringArray decreases its size by one. If the index value is invalid nothing is changed. If the last String reference is removed the StringArray becomes empty. The size of the internal Java array may be decreased by creating a smaller array and copying the values across, depending on how you decide to manage the array. Otherwise, the array is not changed but now has one more unused element.

```
public void remove(int index)
```

- Return true if the StringArray contains a given String, false otherwise. Contains means that the StringArray holds a reference to a String with the same String value, but this is not comparing the references, it is comparing the actual strings held by the String objects. Strings are compared ignoring the upper/lower case of the letters (i.e., 'Hello' and 'hello' are considered the same).

```
public boolean contains(String s)
```

- Return true if the StringArray contains a given String, false otherwise. Contains means that the StringArray holds a reference to a String with the same String value, but this is not comparing the references, it is comparing the actual strings held by the String objects. The upper/lower case of each letter in the strings must match (i.e., ‘Hello’ and ‘hello’ are not considered the same).

```
public boolean containsMatchingCase(String s)
```

- Return the index position of a String in the StringArray, or -1 if the string is not present. The index of the first match is returned. Match means that the StringArray holds a reference to a String with the same String value, but this is not comparing the references, it is comparing the actual strings held by the String objects. Strings are matched ignoring the upper/lower case of the letters (i.e., ‘Hello’ and ‘hello’ are considered the same).

```
public int indexOf(String s)
```

- Return the index position of a String in the StringArray, or -1 if the string is not present. The index of the first match is returned. Match means that the StringArray holds a reference to a String with the same String value, but this is not comparing the references, it is comparing the actual strings held by the String objects. In addition, the upper/lower case of each letter in the strings must match (i.e., ‘Hello’ and ‘hello’ are not considered the same).

```
public int indexOfMatchingCase(String s)
```

Q2. Using classes, write a program that can perform spelling checking on a section of text. You must use your answer to Q1 as your data structure class, you cannot use other classes like ArrayList. You will be using the standard Java String class and can use all of the methods it provides.

The basic strategy is to take a string or sequence of strings containing text, extract each word in turn, check that the word exists in a dictionary of words, and output a list of words that don’t match. Basic matching of identical strings representing words will be sufficient, but punctuation such as full-stops and commas will need to be removed from the text being spell-checked.

For the dictionary use the file that is available on the Moodle site, consisting of a long list of words, one per line. If you have a Linux or MacOS computer this is the file you can usually find at /usr/share/dict/words.

To read input such as a text file and the dictionary file you can use the FileInput class from the Moodle site. You need to add both Input and FileInput to your program. This code illustrates how to read a text file:

```
    StringArray words = new StringArray();
    FileInput input = new FileInput("words");
    while (input.hasNextLine()) {
      String s = input.nextLine();
      words.add(s);
    }
```

Alternatively (this is optional) you can use the standard Java classes or Streams for reading input from files, but you will need to understand how to use exceptions.

The input text and dictionary words will be Strings that must be stored using your StringArray class.

You should decide what classes you need and what methods each class has. All instance variables must be private.

There should also be a class Main that contains the main method with the code to initialise and run the program.

Challenge Question

Q3. Extend your program to show the user a list of possible correct spellings if an incorrectly spelt word is found, so that the preferred spelling can be selected. The corrected text after spell checking should be written to a file.