CSE115 Introduction to Computer Science I
Homework 2, part 1retrospective Fall 2017

In this exercise you were given four methods to define, focusing on transforming values passed in to the methods through parameters to produce return values, often using for loops. There were two versions of this exercise – though the scenario was a little different in both cases the structure of the solution was the same in both.

METHOD 1

The first method was described as follows:

```
/*
* The game will use a dictionary of words.
*
* The 'starter words' are all supposed to be of a certain length.
* In the TextTwist2 game the 'starter words' are all of length 7.
*
* Write a method that takes in an ArrayList<String> called words and
* an int named length and returns a new ArrayList<String> containing
* the members of list that contain exactly length characters.
*
* For example, is dictionary is an ArrayList<String>, then calling
*
*     filterWordsForLength(dictionary, 7)
*
* will produce an ArrayList<String> of seven-letter words.
*
*/
```

Here's the method stub you were provided with:

```
public ArrayList<String> filterWordsForLength(ArrayList<String> words, int length) {
       ArrayList<String> answer = new ArrayList<String>();
       // put your code here
       return answer;
}
```

The basic function of this method is to ***filter*** the input ArrayList to produce another ArrayList consisting of a subset of the values in the original. This is a pattern that we will encounter repeatedly (see exercise 3 retrospective, for example).

The basic approach to writing a filtering method is to write a loop to process each member of the input ArrayList, and to incorporate an if statement that determines whether or not to add a given item to the answer ArrayList.

The provided "stub" creates the ArrayList in which we will accumulate the answer:

```
public ArrayList<String> filterWordsForLength(ArrayList<String> words, int length) {
	ArrayList<String> answer = new ArrayList<String>();




	return answer;
}
```

We add a for loop to process each item of the input `list`:

```
public ArrayList<String> filterWordsForLength(ArrayList<String> words, int length) {
	ArrayList<String> answer = new ArrayList<String>();
	for (int i=0; i<words.size(); i=i+1) {



	}
	return answer;
}
```

Finally, we add a conditional statement that determines whether the String at position i should be added to the ArrayList answer:

```
public ArrayList<String> filterWordsForLength(ArrayList<String> words, int length) {
	ArrayList<String> answer = new ArrayList<String>();
	for (int i=0; i<words.size(); i=i+1) {
		if (words.get(i).length() == length) {
			answer.add(words.get(i));
		}
	}
	return answer;
}
```

In the above example we solved the problem using a traditional for loop. We could use a foreach loop as well:

```
public ArrayList<String> filterWordsForLength(ArrayList<String> words, int length) {
	ArrayList<String> answer = new ArrayList<String>();
	for (String t : words) {
		if (t.length() == length) {
			answer.add(t);
		}
	}
	return answer;
}
```

METHOD 2

The second method was given to you:

```
/*
 * This method accepts a String as input and returns an ArrayList<Character>
 * consisting of the characters from the String word.
 *
 * N.B. Character is a new type for us.  A Character represents a single
 * character from a String.
 *
 * Note that word.charAt(i) returns a value that can be directly added to an
 * ArrayList<Character> using the add method of the ArrayList<Character>.
 *
 * For example, string2charList("Wilma") must yield the ArrayList<Character>
 * that prints as [W, i, l, m, a]
 */
```

The provided "stub" creates the ArrayList in which we will accumulate the answer:

```
public ArrayList<Character> string2charList(String word) {
       ArrayList<Character> list = new ArrayList<Character>();
       // put your code here
       return list;
}
```

This method does not filter the input ArrayList. Instead it simply processes each character from the input String and puts each into the answer ArrayList<Character>. For first set up a loop to process every position within the String word:

```
public ArrayList<Character> string2charList(String word) {
       ArrayList<Character> list = new ArrayList<Character>();
       for (int i=0; i<word.length(); i=i+1) {

       }
       return list;
}
```

In the loop we simply add each Character to list.

```
public ArrayList<Character> string2charList(String word) {
       ArrayList<Character> list = new ArrayList<Character>();
       for (int i=0; i<word.length(); i=i+1) {
              list.add(word.charAt(i));
       }
       return list;
}
```

METHOD 3

The third method was described as follows:

```
/*
* This method determines whether or not a given String is an anagram of some
* subset of the letters in the ArrayList<Character>.
*
* See: http://www.dictionary.com/browse/anagram
*
* The basic idea here is that we'll loop through each character in word, and
* remove each word from the ArrayList<Character>.  The remove method of the
* ArrayList removes ONE occurrence from the list.
*
* Example:  Suppose list is the ArrayList<String> that prints as
* [b, o, o, k, k, e, e, p, e, r] then list.remove('e') changes list to
* [b, o, o, k, k, e, p, e, r].
*
* Calling list.remove('e') again changes list to [b, o, o, k, k, p, e, r].
*
* The remove method returns a boolean value.  If the call changes the contents
* of the ArrayList the method returns true.  If calling the method does not
* change the ArrayList then the method returns false.
*
* HINT: because this method will remove characters from ArrayList<Character> it
* is working with, it is important to make a copy of what's in reference before
* using it.  Write a loop that copies the contents of reference to a new
* ArrayList<Character>.
*/
```

The given stub simply returns false:

```
public boolean anagramOfLetterSubset(String word, ArrayList<Character> reference) {
       // put your code here
       return false;  // change the value returned
}
```

This method was a bit more involved. The first thing I did was handle the hint:

```
* HINT: because this method will remove characters from ArrayList<Character> it
* is working with, it is important to make a copy of what's in reference before
* using it.  Write a loop that copies the contents of reference to a new
* ArrayList<Character>.
```

I first make a new ArrayList<Character>:

```
public boolean anagramOfLetterSubset(String word, ArrayList<Character> reference) {
       ArrayList<Character> copy = new ArrayList<Character>();
       return false;
}
```

Then I wrote a loop to copy each element from reference to copy:

```
public boolean anagramOfLetterSubset(String word, ArrayList<Character> reference) {
	ArrayList<Character> copy = new ArrayList<Character>();
	for (int i=0; i<reference.size(); i=i+1) {
		copy.add(reference.get(i));
	}
	return false;
}
```

Next, I consider each Character in word in turn inside a loop:

```
public boolean anagramOfLetterSubset(String word, ArrayList<Character> reference) {
	ArrayList<Character> copy = new ArrayList<Character>();
	for (int i=0; i<reference.size(); i=i+1) {
		copy.add(reference.get(i));
	}
	for (int i=0; i<word.length(); i=i+1) {
		Character c = word.charAt(i);
	}
	return false;
}
```

I then use the remove method to attempt to remove each Character object from copy, paying attention to the value returned. If remove returns true that means the removal was successful: this mean that the Character was in copy to begin with. If remove returns false that means Character was not in copy. This implies that word is NOT an anagram of the given reference letters. In this case the method can immediately return false.

If the loop completes then all the Characters or word were in copy, so word is an anagram of the given reference letters and the method must return true:

```
public boolean anagramOfLetterSubset(String word, ArrayList<Character> reference) {
	ArrayList<Character> copy = new ArrayList<Character>();
	for (int i=0; i<reference.size(); i=i+1) {
		copy.add(reference.get(i));
	}
	for (int i=0; i<word.length(); i=i+1) {
		Character c = word.charAt(i);
		if (!copy.remove(c)) { // indicates c is NOT in reference
			return false;
		}
	}
	return true;
}
```

METHOD 4

The fourth method was described as follows:

```
	/*
	* This method takes a word (a String) and a dictionary of words (an
	* ArrayList<String>) and returns a collection of words (a HashSet<String>) that
	* are anagrams of some subset of the letters in word.
	*
	* Put another way, this method finds all the words or length at least 2 that
	* can be played from the letters in word.
	*
	* HashSet is a collection that, for our purposes in this homework, behaves like
	* an ArrayList with the following exception: calling add(X) on a HashSet adds X
	* only if X is not already in the collection.  In other words, HashSet does not
	* allow duplicate entries.  Because HashSet does not allow duplicates we get
	* unique words in the result.
	*
	* HINT: in defining this method you should find a natural use for both
	* string2charList and also anagramOfLetterSubset.
	*/
```

The given stub simply returns an empty HashSet<String>:

```
public HashSet<String> possibleWords(String word, ArrayList<String> dictionary) {
	HashSet<String> words = new HashSet<String>();
	// put your code here
	return words;
}
```

This is a filtering problem again: return a HashSet<String> of the Strings from dictionary that are anagrams of word. The basic structure will therefore be an if statement inside a loop. The loop considers each String in dictionary in turn, and the if statement determines whether to add a given String to the HashSet.

First I set up the loop (here I use a foreach loop):

```
public HashSet<String> possibleWords(String word, ArrayList<String> dictionary) {
	HashSet<String> words = new HashSet<String>();

	for (String w : dictionary) {



	}
	return words;
}
```

Next I add the if statement:

```
public HashSet<String> possibleWords(String word, ArrayList<String> dictionary) {
	HashSet<String> words = new HashSet<String>();
```

```
        for (String w : dictionary) {
                if (                                                ) {
                        words.add(w);
                }
        }
        return words;
}
```

This much is pretty standard boilerplate code. The interesting part is now trying to figure out under what conditions to add a String w to the HashSet words. There are two requirements;

1) w must be of length at least 2
2) w must be an anagram of the letters in word

This gets us a little further along:

```
public HashSet<String> possibleWords(String word, ArrayList<String> dictionary) {
        HashSet<String> words = new HashSet<String>();
        ArrayList<Character> reference = string2charList(word);
        for (String w : dictionary) {
                if (w.length() > 2 && anagramOfLetterSubset(              )) {
                        words.add(w);
                }
        }
        return words;
}
```

But how to we call anagramOfLetterSubset? What should the arguments be? To determine this we look at the parameter list of the method:

```
public boolean anagramOfLetterSubset(String word, ArrayList<Character> reference)
```

This tells us that we need to provide a String and an ArrayList<Character>. The String is the word whose status as an anagram is in question, and the ArrayList<Character> is the inventory of letters we have at our disposal.

In our possibleWords method the inventory of letters is given by the String word. We can use string2charList to explode that String into an ArrayList<Character>:

```
public HashSet<String> possibleWords(String word, ArrayList<String> dictionary) {
        HashSet<String> words = new HashSet<String>();
        ArrayList<Character> reference = string2charList(word);
        for (String w : dictionary) {
                if (w.length() > 2 && anagramOfLetterSubset(w, reference)) {
                        words.add(w);
                }
        }
        return words;
}
```