

2. This question involves the `SignedText` class, which contains methods that are used to include a signature as part of a string of text. You will write the complete `SignedText` class, which contains a constructor and two methods.

The `SignedText` constructor takes two `String` parameters. The first parameter is a first name and the second parameter is a last name. The length of the second parameter is always greater than or equal to 1.

The `getSignature` method takes no parameters and returns a formatted signature string constructed from the first and last names according to the following rules.

- If the first name is an empty string, the returned signature string contains just the last name.
- If the first name is not an empty string, the returned signature string is the first letter of the first name, a dash ("–"), and the last name, in that order.

The `addSignature` method returns a possibly revised copy of its `String` parameter. The parameter will contain at most one occurrence of the object's signature, at either the beginning or the end of the parameter. The returned string is created from the parameter according to the following rules.

- If the object's signature does not occur in the `String` parameter of the method, the returned `String` is the value of the parameter with the signature added to the end.
- If the object's signature occurs at the end of the `String` parameter, the returned `String` is the unchanged value of the parameter.
- If the object's signature occurs at the beginning of the `String` parameter, the returned `String` is the value of the original parameter with the signature removed from the beginning and appended to the end of the parameter.

The following table contains a sample code execution sequence and the corresponding results. The code execution sequence appears in a class other than `SignedText`.

Statement	Method Call Return Value (blank if none)	Explanation
<code>SignedText st1 = new SignedText("", "Wong");</code>		The <code>SignedText</code> object <code>st1</code> has an empty first name and last name "Wong".
<code>String temp = st1.getSignature();</code>	"Wong"	
<code>SignedText st2 = new SignedText("henri", "dubois");</code>		The <code>SignedText</code> object <code>st2</code> has first name "henri" and last name "dubois".
<code>temp = st2.getSignature();</code>	"h-dubois"	
<code>SignedText st3 = new SignedText("GRACE", "LOPEZ");</code>		The <code>SignedText</code> object <code>st3</code> has first name "GRACE" and last name "LOPEZ".
<code>temp = st3.getSignature();</code>	"G-LOPEZ"	
<code>SignedText st4 = new SignedText("", "FOX");</code>		The <code>SignedText</code> object <code>st4</code> has an empty first name and last name "FOX".
<code>String text = "Dear";</code>		
<code>temp = st4.addSignature(text);</code>	"DearFOX"	The signature does not occur in the <code>addSignature</code> parameter, so the returned string is the value of the parameter with the signature appended.
<code>text = "Best wishesFOX";</code>		
<code>temp = st4.addSignature(text);</code>	"Best wishesFOX"	The signature occurs at the end of the <code>addSignature</code> parameter, so the returned string is the unchanged value of the parameter.

Statement	Method Call Return Value (blank if none)	Explanation
<code>text = "FOXThanks";</code>		
<code>temp = st4.addSignature(text);</code>	"ThanksFOX"	The signature occurs at the beginning of the <code>addSignature</code> parameter, so the returned string is the value of the original parameter with the signature removed from the beginning and appended to the end of the parameter.
<code>text = "G-LOPEZHello";</code>		
<code>temp = st3.addSignature(text);</code>	"HelloG-LOPEZ"	The signature occurs at the beginning of the <code>addSignature</code> parameter, so the returned string is the value of the original parameter with the signature removed from the beginning and appended to the end of the parameter.

Write the complete `SignedText` class. Your implementation must meet all specifications and conform to the examples in the table.

2. This question involves methods that distribute text across lines of an electronic sign. The electronic sign and the text to be displayed on it are represented by the `Sign` class. You will write the complete `Sign` class, which contains a constructor and two methods.

The `Sign` class constructor has two parameters. The first parameter is a `String` that contains the message to be displayed on the sign. The second parameter is an `int` that contains the *width* of each line of the sign. The width is the positive maximum number of characters that can be displayed on a single line of the sign.

A sign contains as many lines as are necessary to display the entire message. The message is split among the lines of the sign without regard to spaces or punctuation. Only the last line of the sign may contain fewer characters than the width indicated by the constructor parameter.

The following are examples of a message displayed on signs of different widths. Assume that in each example, the sign is declared with the width specified in the first column of the table and with the message "Everything on sale, please come in", which contains 34 characters.

Width of the Sign	Sign Display
15	Everything on s ale, please com e in
17	Everything on sal e, please come in
40	Everything on sale, please come in

In addition to the constructor, the `Sign` class contains two methods.

The `numberOfLines` method returns an `int` representing the number of lines needed to display the text on the sign. In the previous examples, `numberOfLines` would return 3, 2, and 1, respectively, for the sign widths shown in the table.

The `getLines` method returns a `String` containing the message broken into lines separated by semicolons (;) or returns `null` if the message is the empty string. The constructor parameter that contains the message to be displayed will not include any semicolons. As an example, in the first row of the preceding table, `getLines` would return "Everything on s;ale, please com;e in". No semicolon should appear at the end of the `String` returned by `getLines`.

The following table contains a sample code execution sequence and the corresponding results. The code execution sequence appears in a class other than `Sign`.

Statement	Method Call Return Value (blank if none)	Explanation
<code>String str;</code>		
<code>int x;</code>		
<code>Sign sign1 = new Sign("ABC222DE", 3);</code>		The message for <code>sign1</code> contains 8 characters, and the sign has lines of width 3.
<code>x = sign1.numberOfLines();</code>	3	The sign needs three lines to display the 8-character message on a sign with lines of width 3.
<code>str = sign1.getLines();</code>	"ABC;222;DE"	Semicolons separate the text displayed on the first, second, and third lines of the sign.
<code>str = sign1.getLines();</code>	"ABC;222;DE"	Successive calls to <code>getLines</code> return the same value.
<code>Sign sign2 = new Sign("ABCD", 10);</code>		The message for <code>sign2</code> contains 4 characters, and the sign has lines of width 10.
<code>x = sign2.numberOfLines();</code>	1	The sign needs one line to display the 4-character message on a sign with lines of width 10.
<code>str = sign2.getLines();</code>	"ABCD"	No semicolon appears, since the text to be displayed fits on the first line of the sign.
<code>Sign sign3 = new Sign("ABCDEF", 6);</code>		The message for <code>sign3</code> contains 6 characters, and the sign has lines of width 6.
<code>x = sign3.numberOfLines();</code>	1	The sign needs one line to display the 6-character message on a sign with lines of width 6.
<code>str = sign3.getLines();</code>	"ABCDEF"	No semicolon appears, since the text to be displayed fits on the first line of the sign.
<code>Sign sign4 = new Sign("", 4);</code>		The message for <code>sign4</code> is an empty string.
<code>x = sign4.numberOfLines();</code>	0	There is no text to display.
<code>str = sign4.getLines();</code>	null	There is no text to display.
<code>Sign sign5 = new Sign("AB_CD_EF", 2);</code>		The message for <code>sign5</code> contains 8 characters, and the sign has lines of width 2.
<code>x = sign5.numberOfLines();</code>	4	The sign needs four lines to display the 8-character message on a sign with lines of width 2.
<code>str = sign5.getLines();</code>	"AB;_C;D_;EF"	Semicolons separate the text displayed on the four lines of the sign.

Write the complete `Sign` class. Your implementation must meet all specifications and conform to the examples shown in the preceding table.

Begin your response at the top of a new page in the separate Free Response booklet and fill in the appropriate circle at the top of each page to indicate the question number. If there are multiple parts to this question, write the part letter with your response.

1. This question involves the `WordMatch` class, which stores a secret string and provides methods that compare other strings to the secret string. You will write two methods in the `WordMatch` class.

```
public class WordMatch
{
    /** The secret string. */
    private String secret;

    /** Constructs a WordMatch object with the given secret string of lowercase letters. */
    public WordMatch(String word)
    {
        /* implementation not shown */
    }

    /** Returns a score for guess, as described in part (a).
     * Precondition: 0 < guess.length() <= secret.length()
     */
    public int scoreGuess(String guess)
    { /* to be implemented in part (a) */ }

    /** Returns the better of two guesses, as determined by scoreGuess and the rules for a
     * tie-breaker that are described in part (b).
     * Precondition: guess1 and guess2 contain all lowercase letters.
     * guess1 is not the same as guess2.
     */
    public String findBetterGuess(String guess1, String guess2)
    { /* to be implemented in part (b) */ }
}
```

- (a) Write the `WordMatch` method `scoreGuess`. To determine the score to be returned, `scoreGuess` finds the number of times that `guess` occurs as a substring of `secret` and then multiplies that number by the square of the length of `guess`. Occurrences of `guess` may overlap within `secret`.

Assume that the length of `guess` is less than or equal to the length of `secret` and that `guess` is not an empty string.

The following examples show declarations of a `WordMatch` object. The tables show the outcomes of some possible calls to the `scoreGuess` method.

```
WordMatch game = new WordMatch("mississippi");
```

Value of <code>guess</code>	Number of Substring Occurrences	Score Calculation: (Number of Substring Occurrences) x (Square of the Length of <code>guess</code>)	Return Value of <code>game.scoreGuess(guess)</code>
"i"	4	$4 * 1 * 1 = 4$	4
"iss"	2	$2 * 3 * 3 = 18$	18
"issipp"	1	$1 * 6 * 6 = 36$	36
"mississippi"	1	$1 * 11 * 11 = 121$	121

```
WordMatch game = new WordMatch("aaaabb");
```

Value of <code>guess</code>	Number of Substring Occurrences	Score Calculation: (Number of Substring Occurrences) x (Square of the Length of <code>guess</code>)	Return Value of <code>game.scoreGuess(guess)</code>
"a"	4	$4 * 1 * 1 = 4$	4
"aa"	3	$3 * 2 * 2 = 12$	12
"aaa"	2	$2 * 3 * 3 = 18$	18
"aabb"	1	$1 * 4 * 4 = 16$	16
"c"	0	$0 * 1 * 1 = 0$	0

Complete the `scoreGuess` method.

```
/** Returns a score for guess, as described in part (a).
 * Precondition: 0 < guess.length() <= secret.length()
 */
public int scoreGuess(String guess)
```

Begin your response at the top of a new page in the separate Free Response booklet and fill in the appropriate circle at the top of each page to indicate the question number. If there are multiple parts to this question, write the part letter with your response.

Class information for this question

```
public class WordMatch
{
    private String secret

    public WordMatch(String word)
    public int scoreGuess(String guess)
    public String findBetterGuess(String guess1, String guess2)
```

- (b) Write the `WordMatch` method `findBetterGuess`, which returns the better guess of its two `String` parameters, `guess1` and `guess2`. If the `scoreGuess` method returns different values for `guess1` and `guess2`, then the guess with the higher score is returned. If the `scoreGuess` method returns the same value for `guess1` and `guess2`, then the alphabetically greater guess is returned.

The following example shows a declaration of a `WordMatch` object and the outcomes of some possible calls to the `scoreGuess` and `findBetterGuess` methods.

```
WordMatch game = new WordMatch("concatenation");
```

Method Call	Return Value	Explanation
<code>game.scoreGuess("ten");</code>	9	1 * 3 * 3
<code>game.scoreGuess("nation");</code>	36	1 * 6 * 6
<code>game.findBetterGuess("ten", "nation");</code>	"nation"	Since <code>scoreGuess</code> returns 36 for "nation" and 9 for "ten", the guess with the greater score, "nation", is returned.
<code>game.scoreGuess("con");</code>	9	1 * 3 * 3
<code>game.scoreGuess("cat");</code>	9	1 * 3 * 3
<code>game.findBetterGuess("con", "cat");</code>	"con"	Since <code>scoreGuess</code> returns 9 for both "con" and "cat", the alphabetically greater guess, "con", is returned.

Complete method `findBetterGuess`.

Assume that `scoreGuess` works as specified, regardless of what you wrote in part (a). You must use `scoreGuess` appropriately to receive full credit.

```
/** Returns the better of two guesses, as determined by scoreGuess and the rules for a
 * tie-breaker that are described in part (b).
 * Precondition: guess1 and guess2 contain all lowercase letters.
 * guess1 is not the same as guess2.
 */
public String findBetterGuess(String guess1, String guess2)
```

Begin your response at the top of a new page in the separate Free Response booklet and fill in the appropriate circle at the top of each page to indicate the question number. If there are multiple parts to this question, write the part letter with your response.

Class information for this question

```
public class WordMatch
```

```
private String secret
```

```
public WordMatch(String word)
```

```
public int scoreGuess(String guess)
```

```
public String findBetterGuess(String guess1, String guess2)
```

3. The `StringChecker` interface describes classes that check if strings are valid, according to some criterion.

```
public interface StringChecker
{
    /** Returns true if str is valid. */
    boolean isValid(String str);
}
```

A `CodeWordChecker` is a `StringChecker`. A `CodeWordChecker` object can be constructed with three parameters: two integers and a string. The first two parameters specify the minimum and maximum code word lengths, respectively, and the third parameter specifies a string that must not occur in the code word. A `CodeWordChecker` object can also be constructed with a single parameter that specifies a string that must not occur in the code word; in this case the minimum and maximum lengths will default to 6 and 20, respectively.

The following examples illustrate the behavior of `CodeWordChecker` objects.

Example 1

```
StringChecker sc1 = new CodeWordChecker(5, 8, "$");
```

Valid code words have 5 to 8 characters and must not include the string "\$".

Method call	Return value	Explanation
<code>sc1.isValid("happy")</code>	true	The code word is valid.
<code>sc1.isValid("happy\$")</code>	false	The code word contains "\$".
<code>sc1.isValid("Code")</code>	false	The code word is too short.
<code>sc1.isValid("happyCode")</code>	false	The code word is too long.

Example 2

```
StringChecker sc2 = new CodeWordChecker("pass");
```

Valid code words must not include the string "pass". Because the bounds are not specified, the length bounds are 6 and 20, inclusive.

Method call	Return value	Explanation
<code>sc2.isValid("MyPass")</code>	true	The code word is valid.
<code>sc2.isValid("Mypassport")</code>	false	The code word contains "pass".
<code>sc2.isValid("happy")</code>	false	The code word is too short.
<code>sc2.isValid("1,000,000,000,000,000")</code>	false	The code word is too long.

Write the complete `CodeWordChecker` class. Your implementation must meet all specifications and conform to all examples.

3. This question involves analyzing and modifying a string. The following `Phrase` class maintains a phrase in an instance variable and has methods that access and make changes to the phrase. You will write two methods of the `Phrase` class.

```
public class Phrase
{
    private String currentPhrase;

    /** Constructs a new Phrase object. */
    public Phrase(String p)
    {
        currentPhrase = p;
    }

    /** Returns the index of the nth occurrence of str in the current phrase;
     *  returns -1 if the nth occurrence does not exist.
     *  Precondition: str.length() > 0 and n > 0
     *  Postcondition: the current phrase is not modified.
     */
    public int findNthOccurrence(String str, int n)
    {
        /* implementation not shown */
    }

    /** Modifies the current phrase by replacing the nth occurrence of str with repl.
     *  If the nth occurrence does not exist, the current phrase is unchanged.
     *  Precondition: str.length() > 0 and n > 0
     */
    public void replaceNthOccurrence(String str, int n, String repl)
    {
        /* to be implemented in part (a) */
    }

    /** Returns the index of the last occurrence of str in the current phrase;
     *  returns -1 if str is not found.
     *  Precondition: str.length() > 0
     *  Postcondition: the current phrase is not modified.
     */
    public int findLastOccurrence(String str)
    {
        /* to be implemented in part (b) */
    }

    /** Returns a string containing the current phrase. */
    public String toString()
    {
        return currentPhrase;
    }
}
```

Part (a) begins on page 12.

- (a) Write the `Phrase` method `replaceNthOccurrence`, which will replace the `nth` occurrence of the string `str` with the string `repl`. If the `nth` occurrence does not exist, `currentPhrase` remains unchanged.

Several examples of the behavior of the method `replaceNthOccurrence` are shown below.

Code segments	Output produced
<pre>Phrase phrase1 = new Phrase("A cat ate late."); phrase1.replaceNthOccurrence("at", 1, "rane"); System.out.println(phrase1);</pre>	A crane ate late.
<pre>Phrase phrase2 = new Phrase("A cat ate late."); phrase2.replaceNthOccurrence("at", 6, "xx"); System.out.println(phrase2);</pre>	A cat ate late.
<pre>Phrase phrase3 = new Phrase("A cat ate late."); phrase3.replaceNthOccurrence("bat", 2, "xx"); System.out.println(phrase3);</pre>	A cat ate late.
<pre>Phrase phrase4 = new Phrase("aaaa"); phrase4.replaceNthOccurrence("aa", 1, "xx"); System.out.println(phrase4);</pre>	xxaa
<pre>Phrase phrase5 = new Phrase("aaaa"); phrase5.replaceNthOccurrence("aa", 2, "bbb"); System.out.println(phrase5);</pre>	abbba

Class information for this question

```
public class Phrase
{
    private String currentPhrase
    public Phrase(String p)
    public int findNthOccurrence(String str, int n)
    public void replaceNthOccurrence(String str, int n, String repl)
    public int findLastOccurrence(String str)
    public String toString()
}
```

The `Phrase` class includes the method `findNthOccurrence`, which returns the `nth` occurrence of a given string. You must use `findNthOccurrence` appropriately to receive full credit.

Complete method `replaceNthOccurrence` below.

```
/** Modifies the current phrase by replacing the nth occurrence of str with repl.
 * If the nth occurrence does not exist, the current phrase is unchanged.
 * Precondition: str.length() > 0 and n > 0
 */
public void replaceNthOccurrence(String str, int n, String repl)
```

Part (b) begins on page 14.

- (b) Write the `Phrase` method `findLastOccurrence`. This method finds and returns the index of the last occurrence of a given string in `currentPhrase`. If the given string is not found, `-1` is returned. The following tables show several examples of the behavior of the method `findLastOccurrence`.

```
Phrase phrase1 = new Phrase("A cat ate late.");
```

Method call	Value returned
<code>phrase1.findLastOccurrence("at")</code>	11
<code>phrase1.findLastOccurrence("cat")</code>	2
<code>phrase1.findLastOccurrence("bat")</code>	-1

Class information for this question

```
public class Phrase
{
    private String currentPhrase
    public Phrase(String p)
    public int findNthOccurrence(String str, int n)
    public void replaceNthOccurrence(String str, int n, String repl)
    public int findLastOccurrence(String str)
    public String toString()
}
```

WRITE YOUR SOLUTION ON THE NEXT PAGE.

You must use `findNthOccurrence` appropriately to receive full credit.

Complete method `findLastOccurrence` below.

```
/** Returns the index of the last occurrence of str in the current phrase;
 * returns -1 if str is not found.
 * Precondition: str.length() > 0
 * Postcondition: the current phrase is not modified.
 */
```

2. This question involves two classes that are used to process log messages. A list of sample log messages is given below.

```
CLIENT3:security alert - repeated login failures
Webserver:disk offline
SERVER1:file not found
SERVER2:read error on disk DSK1
SERVER1:write error on disk DSK2
Webserver:error on /dev/disk
```

Log messages have the format *machineId:description*, where *machineId* identifies the computer and *description* describes the event being logged. Exactly one colon (":") appears in a log message. There are no blanks either immediately before or immediately after the colon.

The following `LogMessage` class is used to represent a log message.

```
public class LogMessage
{
    private String machineId;
    private String description;

    /** Precondition: message is a valid log message. */
    public LogMessage(String message)
    { /* to be implemented in part (a) */ }

    /** Returns true if the description in this log message properly contains keyword;
     * false otherwise.
     */
    public boolean containsWord(String keyword)
    { /* to be implemented in part (b) */ }

    public String getMachineId()
    { return machineId; }

    public String getDescription()
    { return description; }

    // There may be instance variables, constructors, and methods that are not shown.
}
```

- (a) Write the constructor for the `LogMessage` class. It must initialize the private data of the object so that `getMachineId` returns the *machineId* part of the message and `getDescription` returns the *description* part of the message.

Complete the `LogMessage` constructor below.

```
/** Precondition: message is a valid log message. */  
public LogMessage(String message)
```

Part (b) begins on page 8.

- (b) Write the `LogMessage` method `containsWord`, which returns `true` if the description in the log message *properly contains* a given keyword and returns `false` otherwise.

A description *properly contains* a keyword if all three of the following conditions are true.

- o the keyword is a substring of the description;
- o the keyword is either at the beginning of the description or it is immediately preceded by a space;
- o the keyword is either at the end of the description or it is immediately followed by a space.

The following tables show several examples. The descriptions in the left table properly contain the keyword `"disk"`. The descriptions in the right table do not properly contain the keyword `"disk"`.

Descriptions that properly contain `"disk"`

"disk"
"error on disk"
"error on /dev/disk disk"
"error on disk DSK1 "

Descriptions that do not properly contain `"disk"`

"DISK"
"error on disk3"
"error on /dev/disk"
"diskette"

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Assume that the `LogMessage` constructor works as specified, regardless of what you wrote in part (a).

Complete method `containsWord` below.

```
/** Returns true if the description in this log message properly contains keyword;
 *      false otherwise.
 */
public boolean containsWord(String keyword)
```

Part (c) begins on page 10.

- (c) The `SystemLog` class represents a list of `LogMessage` objects and provides a method that removes and returns a list of all log messages (if any) that properly contain a given keyword. The messages in the returned list appear in the same order in which they originally appeared in the system log. If no message properly contains the keyword, an empty list is returned. The declaration of the `SystemLog` class is shown below.

```
public class SystemLog
{
    /** Contains all the entries in this system log.
     *   Guaranteed not to be null and to contain only non-null entries.
     */
    private List<LogMessage> messageList;

    /** Removes from the system log all entries whose descriptions properly contain keyword,
     *   and returns a list (possibly empty) containing the removed entries.
     *   Postcondition:
     *   - Entries in the returned list properly contain keyword and
     *     are in the order in which they appeared in the system log.
     *   - The remaining entries in the system log do not properly contain keyword and
     *     are in their original order.
     *   - The returned list is empty if no messages properly contain keyword.
     */
    public List<LogMessage> removeMessages(String keyword)
    { /* to be implemented in part (c) */ }

    // There may be instance variables, constructors, and methods that are not shown.
}
```

Write the `SystemLog` method `removeMessages`, which removes from the system log all entries whose descriptions properly contain `keyword` and returns a list of the removed entries in their original order. For example, assume that `theLog` is a `SystemLog` object initially containing six `LogMessage` objects representing the following list of log messages.

```
CLIENT3:security alert - repeated login failures
Webserver:disk offline
SERVER1:file not found
SERVER2:read error on disk DSK1
SERVER1:write error on disk DSK2
Webserver:error on /dev/disk
```

The call `theLog.removeMessages("disk")` would return a list containing the `LogMessage` objects representing the following log messages.

```
Webserver:disk offline
SERVER2:read error on disk DSK1
SERVER1:write error on disk DSK2
```

After the call, `theLog` would contain the following log messages.

```
CLIENT3:security alert - repeated login failures
SERVER1:file not found
Webserver:error on /dev/disk
```

Assume that the `LogMessage` class works as specified, regardless of what you wrote in parts (a) and (b). You must use `containsWord` appropriately to receive full credit.

Complete method `removeMessages` below.

```
/** Removes from the system log all entries whose descriptions properly contain keyword,
 * and returns a list (possibly empty) containing the removed entries.
 * Postcondition:
 * - Entries in the returned list properly contain keyword and
 *   are in the order in which they appeared in the system log.
 * - The remaining entries in the system log do not properly contain keyword and
 *   are in their original order.
 * - The returned list is empty if no messages properly contain keyword.
 */
public List<LogMessage> removeMessages(String keyword)
```

4. This question involves the process of taking a list of words, called `wordList`, and producing a formatted string of a specified length. The list `wordList` contains at least two words, consisting of letters only.

When the formatted string is constructed, spaces are placed in the gaps between words so that as many spaces as possible are evenly distributed to each gap. The equal number of spaces inserted into each gap is referred to as the *basic gap width*. Any *leftover spaces* are inserted one at a time into the gaps from left to right until there are no more leftover spaces.

The following three examples illustrate these concepts. In each example, the list of words is to be placed into a formatted string of length 20.

Example 1: `wordList`: ["AP", "COMP", "SCI", "ROCKS"]

Total number of letters in words: 14

Number of gaps between words: 3

Basic gap width: 2

Leftover spaces: 0

Formatted string:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19																
	A		P					C		O		M		P				S		C		I				R		O		C		K		S	

Example 2: `wordList`: ["GREEN", "EGGS", "AND", "HAM"]

Total number of letters in words: 15

Number of gaps between words: 3

Basic gap width: 1

Leftover spaces: 2

The leftover spaces are inserted one at a time between the words from left to right until there are no more leftover spaces. In this example, the first two gaps get an extra space.

Formatted string:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19																
	G		R		E		E		N				E		G		G		S				A		N		D			H		A		M	

Example 3: `wordList`: ["BEACH", "BALL"]

Total number of letters in words: 9

Number of gaps between words: 1

Basic gap width: 11

Leftover spaces: 0

Formatted string:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19						
	B		E		A		C		H									B		A		L		L	

You will implement three `static` methods in a class named `StringFormatter` that is not shown.

- (a) Write the `StringFormatter` method `totalLetters`, which returns the total number of letters in the words in its parameter `wordList`. For example, if the variable `List<String> words` is `["A", "frog", "is"]`, then the call `StringFormatter.totalLetters(words)` returns 7. You may assume that all words in `wordList` consist of one or more letters.

Complete method `totalLetters` below.

```
/** Returns the total number of letters in wordList.
 * Precondition: wordList contains at least two words, consisting of letters only.
 */
public static int totalLetters(List<String> wordList)
```

Part (b) begins on page 20.

- (b) Write the `StringFormatter` method `basicGapWidth`, which returns the basic gap width as defined earlier.

Class information for this question

```
public class StringFormatter

public static int totalLetters(List<String> wordList)
public static int basicGapWidth(List<String> wordList,
                                int formattedLen)
public static int leftoverSpaces(List<String> wordList,
                                int formattedLen)
public static String format(List<String> wordList, int formattedLen)
```

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Assume that `totalLetters` works as specified regardless of what you wrote in part (a). You must use `totalLetters` appropriately to receive full credit.

Complete method `basicGapWidth` below.

```
/** Returns the basic gap width when wordList is used to produce
 * a formatted string of formattedLen characters.
 * Precondition: wordList contains at least two words, consisting of letters only.
 * formattedLen is large enough for all the words and gaps.
 */
public static int basicGapWidth(List<String> wordList,
                                int formattedLen)
```

Part (c) begins on page 22.

- (c) Write the `StringFormatter` method `format`, which returns the formatted string as defined earlier. The `StringFormatter` class also contains a method called `leftoverSpaces`, which has already been implemented. This method returns the number of leftover spaces as defined earlier and is shown below.

```
/** Returns the number of leftover spaces when wordList is used to produce
 * a formatted string of formattedLen characters.
 * Precondition: wordList contains at least two words, consisting of letters only.
 * formattedLen is large enough for all the words and gaps.
 */
public static int leftoverSpaces(List<String> wordList,
                                int formattedLen)
{ /* implementation not shown */ }
```

Class information for this question

```
public class StringFormatter

public static int totalLetters(List<String> wordList)
public static int basicGapWidth(List<String> wordList,
                                int formattedLen)
public static int leftoverSpaces(List<String> wordList,
                                int formattedLen)
public static String format(List<String> wordList, int formattedLen)
```

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Assume that `basicGapWidth` works as specified, regardless of what you wrote in part (b). You must use `basicGapWidth` and `leftoverSpaces` appropriately to receive full credit.

Complete method `format` below.

```
/** Returns a formatted string consisting of the words in wordList separated by spaces.
 * Precondition: The wordList contains at least two words, consisting of letters only.
 *                 formattedLen is large enough for all the words and gaps.
 * Postcondition: All words in wordList appear in the formatted string.
 * - The words appear in the same order as in wordList.
 * - The number of spaces between words is determined by basicGapWidth and the
 *   distribution of leftoverSpaces from left to right, as described in the question.
 */
public static String format(List<String> wordList, int formattedLen)
```

STOP
END OF EXAM

COMPUTER SCIENCE A
SECTION II
Time—1 hour and 45 minutes
Number of questions—4
Percent of total score—50

Directions: SHOW ALL YOUR WORK. REMEMBER THAT PROGRAM SEGMENTS ARE TO BE WRITTEN IN JAVA.

Notes:

- Assume that the classes listed in the appendices have been imported where appropriate.
 - Unless otherwise noted in the question, assume that parameters in method calls are not `null` and that methods are called only when their preconditions are satisfied.
 - In writing solutions for each question, you may use any of the accessible methods that are listed in classes defined in that question. Writing significant amounts of code that can be replaced by a call to one of these methods may not receive full credit.
1. This question involves reasoning about strings made up of uppercase letters. You will implement two related methods that appear in the same class (not shown). The first method takes a single string parameter and returns a scrambled version of that string. The second method takes a list of strings and modifies the list by scrambling each entry in the list. Any entry that cannot be scrambled is removed from the list.
- (a) Write the method `scrambleWord`, which takes a given word and returns a string that contains a scrambled version of the word according to the following rules.
- The scrambling process begins at the first letter of the word and continues from left to right.
 - If two consecutive letters consist of an "A" followed by a letter that is not an "A", then the two letters are swapped in the resulting string.
 - Once the letters in two adjacent positions have been swapped, neither of those two positions can be involved in a future swap.

The following table shows several examples of words and their scrambled versions.

word	Result returned by <code>scrambleWord(word)</code>
"TAN"	"TNA"
"ABRACADABRA"	"BARCADABARA"
"WHOA"	"WHOA"
"AARDVARK"	"ARADVRAK"
"EGGS"	"EGGS"
"A"	"A"
""	""

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Complete method `scrambleWord` below.

```
/** Scrambles a given word.
 * @param word the word to be scrambled
 * @return the scrambled word (possibly equal to word)
 * Precondition: word is either an empty string or contains only uppercase letters.
 * Postcondition: the string returned was created from word as follows:
 *   - the word was scrambled, beginning at the first letter and continuing from left to right
 *   - two consecutive letters consisting of "A" followed by a letter that was not "A" were swapped
 *   - letters were swapped at most once
 */
public static String scrambleWord(String word)
```

Part (b) begins on page 4.

- (b) Write the method `scrambleOrRemove`, which replaces each word in the parameter `wordList` with its scrambled version and removes any words that are unchanged after scrambling. The relative ordering of the entries in `wordList` remains the same as before the call to `scrambleOrRemove`.

The following example shows how the contents of `wordList` would be modified as a result of calling `scrambleOrRemove`.

Before the call to `scrambleOrRemove`:

	0	1	2	3	4
wordList	"TAN"	"ABRACADABRA"	"WHOA"	"APPLE"	"EGGS"

After the call to `scrambleOrRemove`:

	0	1	2
wordList	"TNA"	"BARCADABARA"	"PAPLE"

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Assume that `scrambleWord` is in the same class as `scrambleOrRemove` and works as specified, regardless of what you wrote in part (a).

Complete method `scrambleOrRemove` below.

```
/** Modifies wordList by replacing each word with its scrambled
 * version, removing any words that are unchanged as a result of scrambling.
 * @param wordList the list of words
 * Precondition: wordList contains only non-null objects
 * Postcondition:
 *   - all words unchanged by scrambling have been removed from wordList
 *   - each of the remaining words has been replaced by its scrambled version
 *   - the relative ordering of the entries in wordList is the same as it was
 *     before the method was called
 */
public static void scrambleOrRemove(List<String> wordList)
```