Prac1

1. This question involves the implementation and extension of a RandomStringChooser class.

a. A RandomStringChooser object is constructed from an array of non-null String values. When the object is first constructed, all of the strings are considered available. The RandomStringChooser class has a getNext method, which has the following behavior. A call to getNext returns a randomly chosen string from the available strings in the object. Once a particular string has been returned from a call to getNext, it is no longer available to be returned from subsequent calls to getNext. If no strings are available to be returned, getNext returns "NONE".

The following code segment shows an example of the behavior of RandomStringChooser.

```
String[] wordArray = {"wheels", "on", "the", "bus"};
RandomStringChooser sChooser = new RandomStringChooser(wordArray);
for (int k = 0; k < 6; k++)
{
   System.out.print(sChooser.getNext() + " ");
}
```

One possible output is shown below. Because sChooser has only four strings, the string "NONE" is printed twice.

```
bus the wheels on NONE NONE
```

Write the entire RandomStringChooser class. Your implementation must include an appropriate constructor and any necessary methods. Any instance variables must be private. The code segment in the example above should have the indicated behavior (that is, it must compile and produce a result like the possible output shown). Neither the constructor nor any of the methods should alter the parameter passed to the constructor, but your implementation may copy the contents of the array.

b. The following partially completed RandomLetterChooser class is a subclass of the RandomStringChooser class. You will write the constructor for the RandomLetterChooser class.

```
public class RandomLetterChooser extends RandomStringChooser
{
    /** Constructs a random letter chooser using the given string str.
     *  Precondition: str contains only letters.
     */
    public RandomLetterChooser(String str)
    {  /* to be implemented in part (b) */  }

    /** Returns an array of single-letter strings.
     *  Each of these strings consists of a single letter from str. Element k
     *  of the returned array contains the single letter at position k of str.
     *  For example, getSingleLetters("cat") returns the
     *  array { "c", "a", "t" }.
     */
    public static String[] getSingleLetters(String str)
    {  /* implementation not shown */  }
}
```

The following code segment shows an example of using RandomLetterChooser.

```
RandomLetterChooser letterChooser = new RandomLetterChooser("cat");
for (int k = 0; k < 4; k++)
{
   System.out.print(letterChooser.getNext());
}
```

The code segment will print the three letters in "cat" in one of the possible orders. Because there are only three letters in the original string, the code segment prints "NONE" the fourth time through the loop. One possible output is shown below.

```
actNONE
```

Assume that the RandomStringChooser class that you wrote in part (a) has been implemented correctly and that getSingleLetters works as specified. You must use getSingleLetters appropriately to receive full credit.

Complete the RandomLetterChooser constructor below.

```
/** Constructs a random letter chooser using the given string str.
 *  Precondition: str contains only letters.
 */
public RandomLetterChooser(String str)
```

---

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)
```

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.

- the keyword is a substring of the description;
- the keyword is either at the beginning of the description or it is immediately preceded by a space;
- 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"`

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)
```

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)
```

---

3. A crossword puzzle grid is a two-dimensional rectangular array of black and white squares. Some of the white squares are labeled with a positive number according to the *crossword labeling rule*.

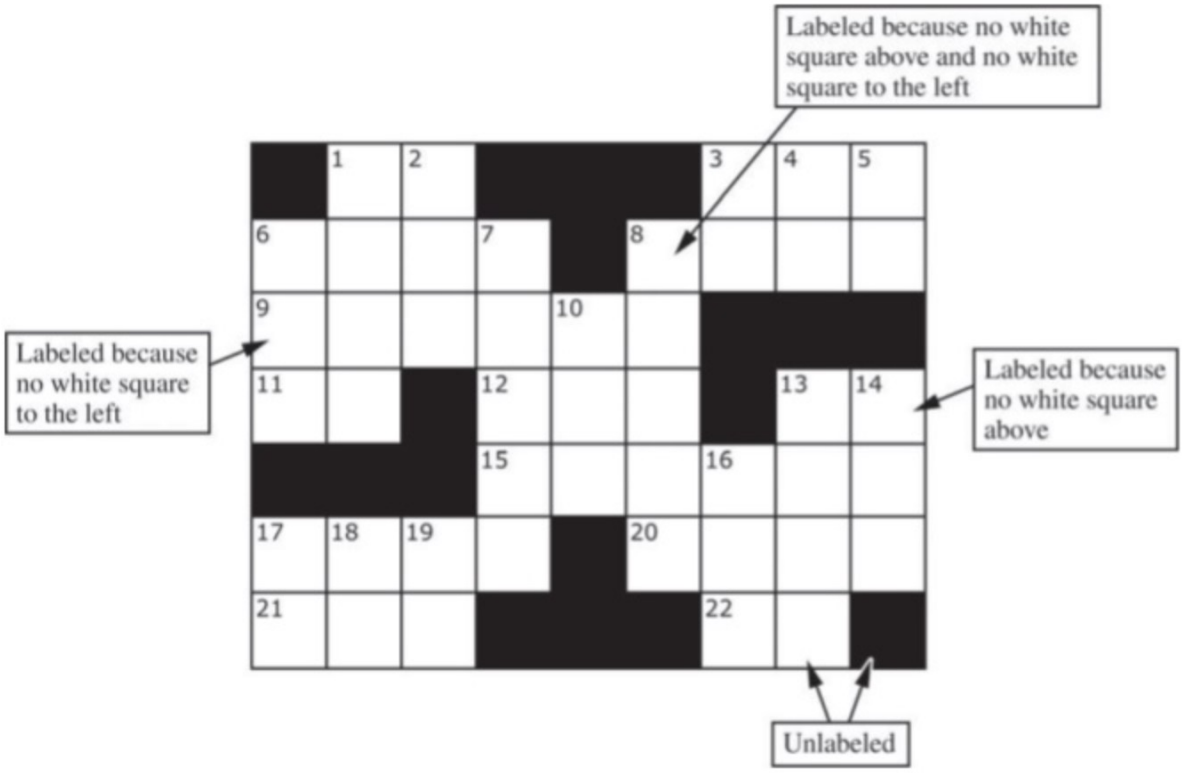
The crossword labeling rule identifies squares to be labeled with a positive number as follows.

A square is labeled with a positive number if and only if

- the square is white and
- the square does not have a white square immediately above it, or it does not have a white square immediately to its left, or both.

The squares identified by these criteria are labeled with consecutive numbers in row-major order, starting at 1.

The following diagram shows a crossword puzzle grid and the labeling of the squares according to the crossword labeling rule.



This question uses two classes, a Square class that represents an individual square in the puzzle and a Crossword class that represents a crossword puzzle grid. A partial declaration of the Square class is shown below.

```
public class Square
{
    /** Constructs one square of a crossword puzzle grid.
     *  Postcondition:
     *    - The square is black if and only if isBlack is true.
     *    - The square has number num.
     */
    public Square(boolean isBlack, int num)
    {  /* implementation not shown */  }

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

A partial declaration of the Crossword class is shown below. You will implement one method and the constructor in the Crossword class.

```
public class Crossword
{
    /** Each element is a Square object with a color (black or white) and a number.
     *  puzzle[r][c] represents the square in row r, column c.
     *  There is at least one row in the puzzle.
     */
    private Square[][] puzzle;

    /** Constructs a crossword puzzle grid.
     *  Precondition: There is at least one row in blackSquares.
     *  Postcondition:
     *    - The crossword puzzle grid has the same dimensions as blackSquares.
     *    - The Square object at row r, column c in the crossword puzzle grid is black
     *      if and only if blackSquares[r][c] is true.
     *    - The squares in the puzzle are labeled according to the crossword labeling rule.
     */
    public Crossword(boolean[][] blackSquares)
    {  /* to be implemented in part (b) */  }

    /** Returns true if the square at row r, column c should be labeled with a positive number;
     *          false otherwise.
     *  The square at row r, column c is black if and only if blackSquares[r][c] is true.
     *  Precondition: r and c are valid indexes in blackSquares.
     */
    private boolean toBeLabeled(int r, int c, boolean[][] blackSquares)
    {  /* to be implemented in part (a) */  }

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

a. Write the Crossword method toBeLabeled. The method returns true if the square indexed by row r, column c in a crossword puzzle grid should be labeled with a positive number according to the crossword labeling rule; otherwise it returns false. The parameter blackSquares indicates which squares in the crossword puzzle grid are black.

Class information for this question

```
public class Square

public Square(boolean isBlack, int num)

public class Crossword

private Square[][] puzzle

public Crossword(boolean[][] blackSquares)
private boolean toBeLabeled(int r, int c, boolean[][] blackSquares)
```

Complete method toBeLabeled below.

```
/** Returns true if the square at row r, column c should be labeled with a positive number;
 *          false otherwise.
 *  The square at row r, column c is black if and only if blackSquares[r][c] is true.
 *  Precondition: r and c are valid indexes in blackSquares.
 */
private boolean toBeLabeled(int r, int c, boolean[][] blackSquares)
```

b. Write the Crossword constructor. The constructor should initialize the crossword puzzle grid to have the same dimensions as the parameter blackSquares. Each element of the puzzle grid should be initialized with a reference to a Square object with the appropriate color and number. The number is positive if the square is labeled and 0 if the square is not labeled.

Class information for this question

```
public class Square

public Square(boolean isBlack, int num)

public class Crossword

private Square[][] puzzle

public Crossword(boolean[][] blackSquares)
private boolean toBeLabeled(int r, int c, boolean[][] blackSquares)
```

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

Complete the Crossword constructor below.

```
/** Constructs a crossword puzzle grid.
 *  Precondition: There is at least one row in blackSquares.
 *  Postcondition:
 *    - The crossword puzzle grid has the same dimensions as blackSquares.
 *    - The Square object at row r, column c in the crossword puzzle grid is black
 *      if and only if blackSquares[r][c] is true.
 *    - The squares in the puzzle are labeled according to the crossword labeling rule.
 */
public Crossword(boolean[][] blackSquares)
```

---

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 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)
```

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)
```

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)
```

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)
```

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.

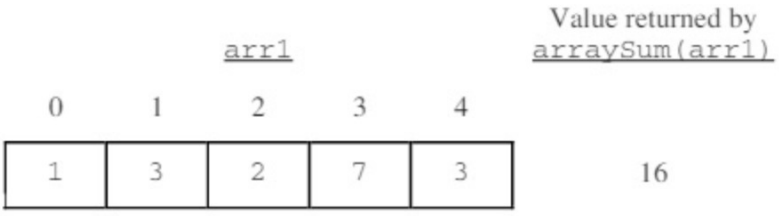
Complete method format below.

```
/** 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)
```

---

5. This question involves reasoning about one-dimensional and two-dimensional arrays of integers. You will write three static methods, all of which are in a single enclosing class, named DiverseArray (not shown). The first method returns the sum of the values of a one-dimensional array; the second method returns an array that represents the sums of the rows of a two-dimensional array; and the third method analyzes row sums.

(a) Write a static method arraySum that calculates and returns the sum of the entries in a specified one-dimensional array. The following example shows an array arr1 and the value returned by a call to arraySum.



Complete method arraySum below.

/ * * Returns the sum of the entries in the one-dimensional array arr.

* /

public static int arraySum (int [ ] arr)

(b) Write a static method rowSums that calculates the sums of each of the rows in a given two-dimensional array and returns these sums in a one-dimensional array. The method has one parameter, a two-dimensional array arr2D of int values. The array is in row-major order: arr2D [r] [c] is the entry at row r and column c. The method returns a one-dimensional array with one entry for each row of arr2D such that each entry is the sum of the corresponding row in arr2D. As a reminder, each row of a two-dimensional array is a one-dimensional array.

For example, if mat1 is the array represented by the following table, the call rowSums(mat1) returns the array {16, 32, 28, 20}.

mat1

	0	1	2	3	4
0	1	3	2	7	3
1	10	10	4	6	2
2	5	3	5	9	6
3	7	6	4	2	1

Methods written in this question

```
public static int arraySum(int[] arr)
public static int[] rowSums(int[][] arr2D)
public static boolean isDiverse(int[][] arr2D)
```

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

Complete method rowSums below.

/ * * Returns a one-dimensional array in which the entry at index k is the sum of

* the entries of row k of the two-dimensional array arr2D.

* /

public static int [ ] rowSums(int [ ] [ ] arr2D)

(c) A two-dimensional array is diverse if no two of its rows have entries that sum to the same value. In the following examples, the array mat1 is diverse because each row sum is different, but the array mat2 is not diverse because the first and last rows have the same sum.

mat1

	0	1	2	3	4	Row sums
0	1	3	2	7	3	16
1	10	10	4	6	2	32
2	5	3	5	9	6	28
3	7	6	4	2	1	20

mat2

	0	1	2	3	4	Row sums
0	1	1	5	3	4	14
1	12	7	6	1	9	35
2	8	11	10	2	5	36
3	3	2	3	0	6	14

Write a static method isDiverse that determines whether or not a given two-dimensional array is diverse. The method has one parameter: a two-dimensional array arr2D of int values. The method should return true if all the row sums in the given array are unique; otherwise, it should return false. In the arrays shown above, the call isDiverse (mat1) returns true and the call isDiverse(mat2) returns false.

Methods written in this question

```
public static int arraySum(int[] arr)
public static int[] rowSums(int[][] arr2D)
public static boolean isDiverse(int[][] arr2D)
```

Assume that arraySum and rowSums work as specified, regardless of what you wrote in parts (a) and(b). You must use rowSums appropriately to receive full credit.
Complete method isDiverse below.

/ * * Returns true if all rows in arr2D have different row sums;
* false otherwise.

* /public static boolean isDiverse(int [ ] [ ] arr2D)

---

6. Consider a guessing game in which a player tries to guess a hidden word. The hidden word contains only capital letters and has a length known to the player. A guess contains only capital letters and has the same length as the hidden word.

After a guess is made, the player is given a hint that is based on a comparison between the hidden word and the guess. Each position in the hint contains a character that corresponds to the letter in the same position in the guess. The following rules determine the characters that appear in the hint.

If the letter in the guess is …	the corresponding character in the hint is
also in the same position in the hidden word,	the matching letter
also in the hidden word, but in a different position,	"+"
not in the hidden word,	"*"

The HiddenWord class will be used to represent the hidden word in the game. The hidden word is passed to the constructor. The class contains a method, getHint, that takes a guess and produces a

hint.

For example, suppose the variable puzzle is declared as follows.

HiddenWord puzzle = new HiddenWord("HARPS");

The following table shows several guesses and the hints that would be produced.

Call to `getHint`	String returned
`puzzle.getHint("AAAAA")`	`"+A+++"`
`puzzle.getHint("HELLO")`	`"H****"`
`puzzle.getHint("HEART")`	`"H*++*"`
`puzzle.getHint("HARMS")`	`"HAR*S"`
`puzzle.getHint("HARPS")`	`"HARPS"`

Write the complete HiddenWord class, including any necessary instance variables, its constructor, and the method, getHint, described above. You may assume that the length of the guess is the same as the length of the hidden word.

---

7. This question involves the design of an interface, writing a class that implements the interface, and writing a method that uses the interface.

(a) A *number group* represents a group of integers defined in some way. It could be empty, or it could contain one or more integers.

Write an interface named NumberGroup that represents a group of integers. The interface should have a single contains method that determines if a given integer is in the group. For example, if group1 is of type NumberGroup, and it contains only the two numbers -5 and 3, then group1.contains(-5) would return true, and group1.contains(2) would return false.
Write the complete NumberGroup interface. It must have exactly one method.

(b) A *range* represents a number group that contains all (and only) the integers between a minimum value and a maximum value, inclusive.
Write the Range class, which is a NumberGroup. The Range class represents the group of int values that range from a given minimum value up through a given maximum value, inclusive. For example,the declaration

---

NumberGroup range1 = new Range(-3, 2);

represents the group of integer values -3, -2, -1, 0, 1, 2.

Write the complete Range class. Include all necessary instance variables and methods as well as a constructor that takes two int parameters. The first parameter represents the minimum value, and the second parameter represents the maximum value of the range. You may assume that the minimum is less than or equal to the maximum.

(c) The MultipleGroups class (not shown) represents a collection of NumberGroup objects and isa NumberGroup. The MultipleGroups class stores the number groups in the instance variable groupList (shown below), which is initialized in the constructor.

private List<NumberGroup> groupList;

Write the MultipleGroups method contains. The method takes an integer and returns true if and only if the integer is contained in one or more of the number groups in groupList.

For example, suppose multiple1 has been declared as an instance of MultipleGroups and consists of the three ranges created by the calls new Range(5, 8), new Range(10, 12), and new Range(1, 6). The following table shows the results of several calls to contains.

Call	Result
`multiple1.contains(2)`	`true`
`multiple1.contains(9)`	`false`
`multiple1.contains(6)`	`true`

Complete method `contains` below.

```
/** Returns true if at least one of the number groups in this multiple group contains num;
 *          false otherwise.
 */
public boolean contains(int num)
```

---

8. 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 `scrambleWord(word)`
"TAN"	"TNA"
"ABRACADABRA"	"BARCADABARA"
"WHOA"	"WHOA"
"AARDVARK"	"ARADVRAK"
"EGGS"	"EGGS"
"A"	"A"
""	""

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)
```

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"

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)
```

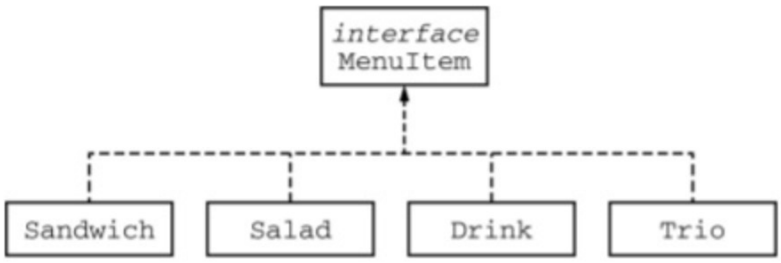
---

9. The menu at a lunch counter includes a variety of sandwiches, salads, and drinks. The menu also allows a customer to create a "trio," which consists of three menu items: a sandwich, a salad, and a drink. The price of the trio is the sum of the two highest-priced menu items in the trio; one item with the lowest price is free.

Each menu item has a name and a price. The four types of menu items are represented by the four classes Sandwich, Salad, Drink, and Trio. All four classes implement the following MenuItem interface.

```
public interface MenuItem
{
  /** @return the name of the menu item */
  String getName();

  /** @return the price of the menu item */
  double getPrice();
}
```

The following diagram shows the relationship between the MenuItem interface and the Sandwich, Salad, Drink, and Trio classes.



For example, assume that the menu includes the following items. The objects listed under each heading are instances of the class indicated by the heading.

Sandwich	Salad	Drink
"Cheeseburger" 2.75	"Spinach Salad" 1.25	"Orange Soda" 1.25
"Club Sandwich" 2.75	"Coleslaw" 1.25	"Cappuccino" 3.50

The menu allows customers to create Trio menu items, each of which includes a sandwich, a salad, and a drink. The name of the Trio consists of the names of the sandwich, salad, and drink, in that order, each separated by "/" and followed by a space and then "Trio". The price of the Trio is the sum of the two highest-priced items in the Trio; one item with the lowest price is free.

A trio consisting of a cheeseburger, spinach salad, and an orange soda would have the name "Cheeseburger/Spinach Salad/Orange Soda Trio" and a price of $4.00 (the two highest prices are $2.75 and $1.25). Similarly, a trio consisting of a club sandwich, coleslaw, and a cappuccino would have the name "Club Sandwich/Coleslaw/Cappuccino Trio" and a price of $6.25 (the two highest prices are $2.75 and $3.50).

Write the Trio class that implements the MenuItem interface. Your implementation must include a constructor that takes three parameters representing a sandwich, salad, and drink. The following code segment should have the indicated behavior.

```
Sandwich sandwich;
Salad salad;
Drink drink;
/* Code that initializes sandwich, salad, and drink */

Trio trio = new Trio(sandwich, salad, drink);   // Compiles without error

Trio trio1 = new Trio(salad, sandwich, drink); // Compile-time error
Trio trio2 = new Trio(sandwich, salad, salad); // Compile-time error
```

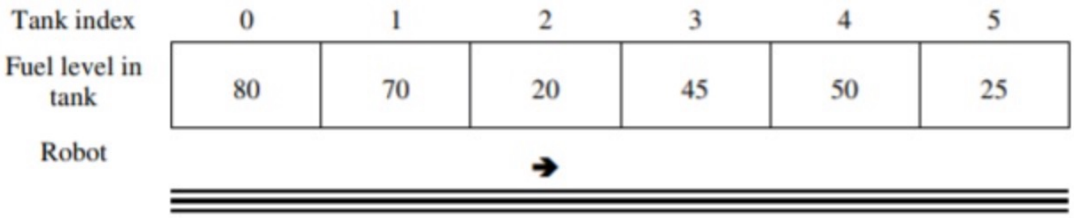
Write the complete Trio class below.

---

10. A fuel depot has a number of fuel tanks arranged in a line and a robot that moves a filling mechanism back and forth along the line so that the tanks can be filled. A fuel tank is specified by the FuelTank interface below.

```
public interface FuelTank
{
  /** @return an integer value that ranges from 0 (empty) to 100 (full) */
  int getFuelLevel();
}
```

A fuel depot keeps track of the fuel tanks and the robot. The following figure represents the tanks and the robot in a fuel depot. The robot, indicated by the arrow, is currently at index 2 and is facing to the right.



The state of the robot includes the index of its location and the direction in which it is facing (to the right or to the left). This information is specified in the FuelRobot interface as shown in the following declaration.

```
public interface FuelRobot
{
  /** @return the index of the current location of the robot */
  int getCurrentIndex();

  /** Determine whether the robot is currently facing to the right
   *  @return true  if the robot is facing to the right (toward tanks with larger indexes)
   *          false if the robot is facing to the left (toward tanks with smaller indexes)
   */
  boolean isFacingRight();

  /** Changes the current direction of the robot */
  void changeDirection();

  /** Moves the robot in its current direction by the number of locations specified.
   *  @param numLocs the number of locations to move. A value of 1 moves
   *                 the robot to the next location in the current direction.
   *         Precondition: numLocs > 0
   */
  void moveForward(int numLocs);
}
```

A fuel depot is represented by the FuelDepot class as shown in the following class declaration.

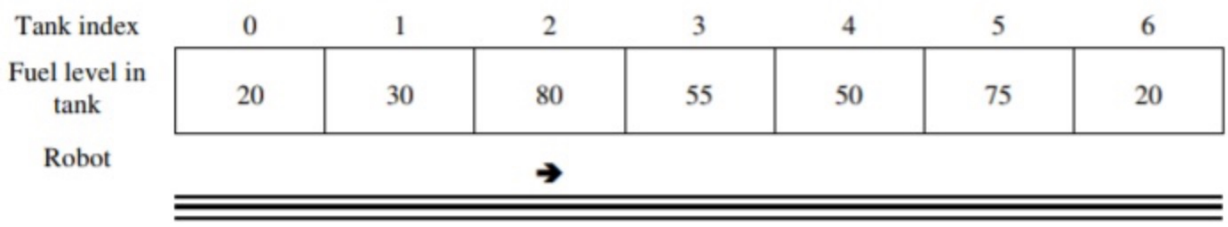
```
public class FuelDepot
{
  /** The robot used to move the filling mechanism */
  private FuelRobot filler;

  /** The list of fuel tanks */
  private List<FuelTank> tanks;

  /** Determines and returns the index of the next tank to be filled.
   *  @param threshold fuel tanks with a fuel level ≤ threshold may be filled
   *  @return index of the location of the next tank to be filled
   *  Postcondition: the state of the robot has not changed
   */
  public int nextTankToFill(int threshold)
  {  /* to be implemented in part (a) */  }

  /** Moves the robot to location locIndex.
   *  @param locIndex the index of the location of the tank to move to
   *         Precondition: 0 ≤ locIndex < tanks.size()
   *  Postcondition: the current location of the robot is locIndex
   */
  public void moveToLocation(int locIndex)
  {  /* to be implemented in part (b) */  }

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

a. Write the FuelDepot method nextTankToFill that returns the index of the next tank to be filled.

The index for the next tank to be filled is determined according to the following rules:

- Return the index of a tank with the lowest fuel level that is less than or equal to a given threshold. If there is more than one fuel tank with the same lowest fuel level, any of their indexes can be returned.
- If there are no tanks with a fuel level less than or equal to the threshold, return the robot's current index.

For example, suppose the tanks contain the fuel levels shown in the following figure.



The following table shows the results of several independent calls to nextTankToFill.

threshold	Return Value	Rationale
50	0 or 6	20 is the lowest fuel level, so either 0 or 6 can be returned.
15	2	There are no tanks with a fuel level ≤ threshold, so the robot's current index is returned.

Complete method nextTankToFill below.

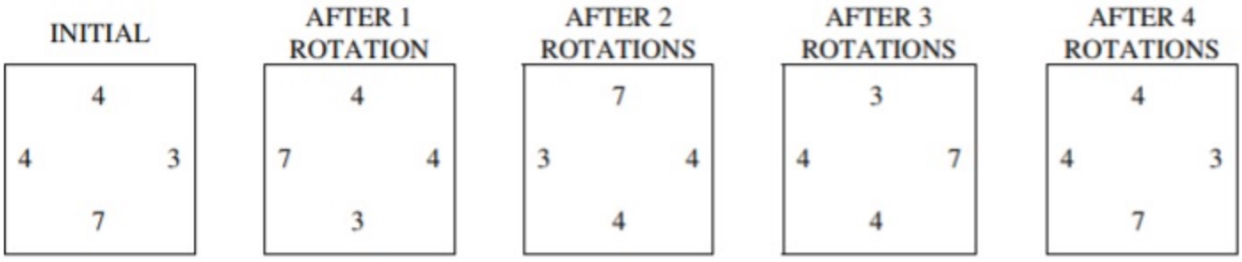
```
/** Determines and returns the index of the next tank to be filled.
 *  @param threshold fuel tanks with a fuel level ≤ threshold may be filled
 *  @return index of the location of the next tank to be filled
 *  Postcondition: the state of the robot has not changed
 */
public int nextTankToFill(int threshold)
```

b. Write the FuelDepot method moveToLocation that will move the robot to the given tank location. Because the robot can only move forward, it may be necessary to change the direction of the robot before having it move. Do not move the robot past the end of the line of fuel tanks. Complete method moveToLocation below.

```
/** Moves the robot to location locIndex.
 *  @param locIndex the index of the location of the tank to move to
 *          Precondition: 0 ≤ locIndex < tanks.size()
 *  Postcondition: the current location of the robot is locIndex
 */
public void moveToLocation(int locIndex)
```

---

11. A game uses square tiles that have numbers on their sides. Each tile is labeled with a number on each of its four sides and may be rotated clockwise, as illustrated below.



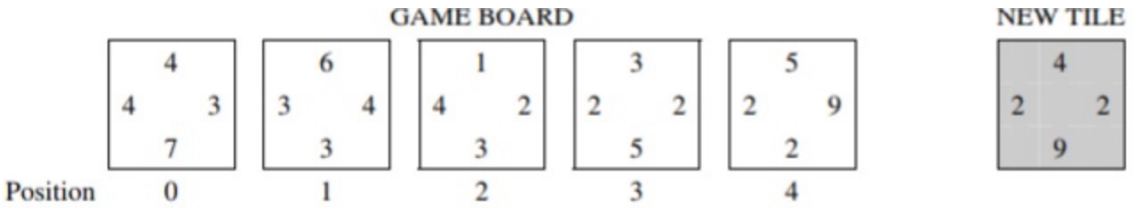
The tiles are represented by the NumberTile class, as given below.

```
public class NumberTile
{
  /** Rotates the tile 90 degrees clockwise
   */
  public void rotate()
  {  /* implementation not shown */  }

  /** @return value at left edge of tile
   */
  public int getLeft()
  {  /* implementation not shown */  }

  /** @return value at right edge of tile
   */
  public int getRight()
  {  /* implementation not shown */  }

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

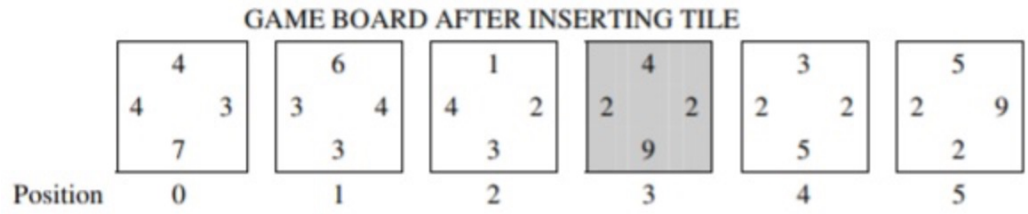
Tiles are placed on a game board so that the adjoining sides of adjacent tiles have the same number. The following figure illustrates an arrangement of tiles and shows a new tile that is to be placed on the

---

game board.



In its original orientation, the new tile can be inserted between the tiles at positions 2 and 3 or between the tiles at positions 3 and 4. If the new tile is rotated once, it can be inserted before the tile at position 0 (the first tile) or after the tile at position 4 (the last tile). Assume that the new tile, in its original orientation, is inserted between the tiles at positions 2 and 3. As a result of the insertion, the tiles at positions 3 and 4 are moved one location to the right, and the new tile is inserted at position 3, as shown below.

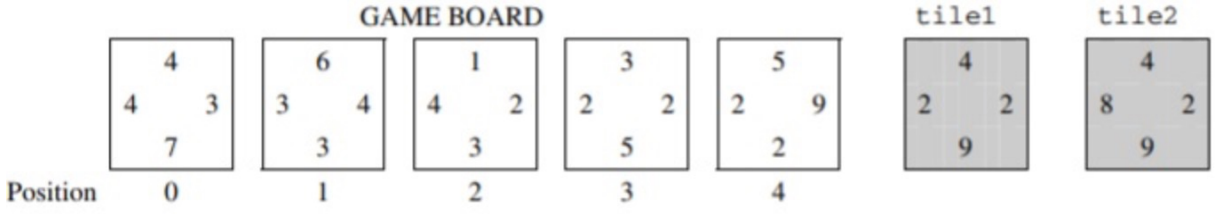


A partial definition of the TileGame class is given below.

```
public class TileGame
{
  /** represents the game board; guaranteed never to be null */
  private ArrayList<NumberTile> board;

  public TileGame()
  { board = new ArrayList<NumberTile>(); }

  /** Determines where to insert tile, in its current orientation, into game board
   *  @param tile the tile to be placed on the game board
   *  @return the position of tile where tile is to be inserted:
   *            0 if the board is empty;
   *           -1 if tile does not fit in front, at end, or between any existing tiles;
   *           otherwise, 0 ≤ position returned ≤ board.size()
   */
  private int getIndexForFit(NumberTile tile)
  {  /* to be implemented in part (a) */  }

  /** Places tile on the game board if it fits (checking all possible tile orientations if necessary).
   *  If there are no tiles on the game board, the tile is placed at position 0.
   *  The tile should be placed at most 1 time.
   *  Precondition: board is not null
   *  @param tile the tile to be placed on the game board
   *  @return true if tile is placed successfully; false otherwise
   *  Postcondition: the orientations of the other tiles on the board are not changed
   *  Postcondition: the order of the other tiles on the board relative to each other is not changed
   */
  public boolean insertTile(NumberTile tile)
  {  /* to be implemented in part (b) */  }

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

a. Write the TileGame method getIndexForFit that determines where a given tile, in its current orientation, fits on the game board. A tile can be inserted at either end of a game board or between two existing tiles if the side(s) of the new tile match the adjacent side(s) of the tile(s) currently on the game board. If there are no tiles on the game board, the position for the insert is 0. The method returns the position that the new tile will occupy on the game board after it has been inserted. If there are multiple possible positions for the tile, the method will return any one of them. If the given tile does not fit anywhere on the game board, the method returns -1.

For example, the following diagram shows a game board and two potential tiles to be placed. The call getIndexForFit(tile1) can return either 3 or 4 because tile1 can be inserted between the tiles at positions 2 and 3, or between the tiles at positions 3 and 4. The call getIndexForFit(tile2) returns -1 because tile2, in its current orientation, does not fit anywhere on the game board.



Complete method getIndexForFit below.

```
/** Determines where to insert tile, in its current orientation, into game board
 *  @param tile the tile to be placed on the game board
 *  @return the position of tile where tile is to be inserted:
 *            0 if the board is empty;
 *           -1 if tile does not fit in front, at end, or between any existing tiles;
 *           otherwise, 0 ≤ position returned ≤ board.size()
 */
private int getIndexForFit(NumberTile tile)
```

b. Write the TileGame method insertTile that attempts to insert the given tile on the game board. The method returns true if the tile is inserted successfully and false only if the tile cannot be placed on the board in any orientation.

Assume that getIndexForFit works as specified, regardless of what you wrote in part (a).

Complete method insertTile below.

```
/** Places tile on the game board if it fits (checking all possible tile orientations if necessary).
 *  If there are no tiles on the game board, the tile is placed at position 0.
 *  The tile should be placed at most 1 time.
 *  Precondition: board is not null
 *  @param tile the tile to be placed on the game board
 *  @return true if tile is placed successfully; false otherwise
 *  Postcondition: the orientations of the other tiles on the board are not changed
 *  Postcondition: the order of the other tiles on the board relative to each other is not changed
 */
public boolean insertTile(NumberTile tile)
```

---

12. In this question, you will implement two methods for a class Hotel that is part of a hotel reservation system. The Hotel class uses the Reservation class shown below. A Reservation is for the person and room number specified when the Reservation is constructed.

```
public class Reservation
{
  public Reservation(String guestName, int roomNumber)
  { /* implementation not shown */ }

  public int getRoomNumber()
  { /* implementation not shown */ }

  // private data and other methods not shown
}
```

An incomplete declaration for the Hotel class is shown below. Each hotel in the hotel reservation system has rooms numbered 0, 1, 2, . . . , up to the last room number in the hotel. For example, a hotel with 100 rooms would have rooms numbered 0, 1, 2, . . . , 99.

```
public class Hotel
{
  private Reservation[] rooms;
    // each element corresponds to a room in the hotel;
    // if rooms[index] is null, the room is empty;
    // otherwise, it contains a reference to the Reservation
    // for that room, such that
    // rooms[index].getRoomNumber() returns index

  private ArrayList waitList;
    // contains names of guests who have not yet been
    // assigned a room because all rooms are full


  // if there are any empty rooms (rooms with no reservation),
  // then create a reservation for an empty room for the
  // specified guest and return the new Reservation;
  // otherwise, add the guest to the end of waitList
  // and return null
  public Reservation requestRoom(String guestName)
  { /* to be implemented in part (a) */ }


  // release the room associated with parameter res, effectively
  // canceling the reservation;
  // if any names are stored in waitList, remove the first name
  // and create a Reservation for this person in the room
  // reserved by res; return that new Reservation;
  // if waitList is empty, mark the room specified by res as empty and
  // return null
  // precondition:  res is a valid Reservation for some room
  //                in this hotel
  public Reservation cancelAndReassign(Reservation res)
  { /* to be implemented in part (b) */ }

  // constructors and other methods not shown
}
```

a. Write the Hotel method requestRoom. Method requestRoom attempts to reserve a room in the hotel for a given guest. If there are any empty rooms in the hotel, one of them will be assigned to the named guest and the newly created reservation is returned. If there are no empty rooms, the guest is added to the end of the waiting list and null is returned.

   Complete method requestRoom below.

```
// if there are any empty rooms (rooms with no reservation),
// then create a reservation for an empty room for the
// specified guest and return the new Reservation;
// otherwise, add the guest to the end of waitList
// and return null
public Reservation requestRoom(String guestName)
```

b. Write the Hotel method cancelAndReassign. Method cancelAndReassign releases a previous reservation. If the waiting list for the hotel contains any names, the vacated room is reassigned to

the first person at the beginning of the list. That person is then removed from the waiting list and the newly created reservation is returned. If no one is waiting, the room is marked as empty and null is returned.

In writing cancelAndReassign you may call any accessible methods in the Reservation and Hotel classes. Assume that these methods work as specified.

Complete method cancelAndReassign below.

```
// release the room associated with parameter res, effectively
// canceling the reservation;
// if any names are stored in waitList, remove the first name
// and create a Reservation for this person in the room
// reserved by res; return that new Reservation;
// if waitList is empty, mark the room specified by res as empty and
// return null
// precondition:  res is a valid Reservation for some room
//                in this hotel
public Reservation cancelAndReassign(Reservation res)
```