Copy of Semester1 Final QuestionBank-FRQ

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

Assume that the classes listed in the Java Quick Reference 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 will not receive full credit.

This question involves the implementation of the `AddtionPattern` class, which generates a number pattern. The `AdditionPattern` object is constructed with two positive integer parameters, as described below.

The first positive integer parameter indicates the starting number in the pattern.
The second positive integer parameter indicates the value that is to be added to obtain each subsequent number in the pattern.

The `AdditionPattern` class also supports the following methods.

`currentNumber`, which returns the current number in the pattern
`next`, which moves to the next number in the pattern
`prev`, which moves to the previous number in the pattern or takes no action if there is no previous number

The following table illustrates the behavior of an `AdditionPattern` object that is instantiated by the following statement.

```
AdditionPattern plus3 = new AdditionPattern(2, 3);
```

Copy of Semester1 Final QuestionBank-FRQ

Method Call	Value Returned (blank if no value)	Explanation
`plus3.currentNumber();`	`2`	The current number is initially the starting number in the pattern.
`plus3.next();`		The pattern adds `3` each time, so move to the next number in the pattern (`5`).
`plus3.currentNumber();`	`5`	The current number is `5`.
`plus3.next();`		The pattern adds `3` each time, so move to the next number in the pattern (`8`).
`plus3.next();`		The pattern adds `3` each time, so move to the next number in the pattern (`11`).
`plus3.currentNumber();`	`11`	The current number is `11`.
`plus3.prev();`		Move to the previous number in the pattern (`8`).
`plus3.prev();`		Move to the previous number in the pattern (`5`).
`plus3.prev();`		Move to the previous number in the pattern (`2`).
`plus3.currentNumber();`	`2`	The current number is `2`.
`plus3.prev();`		There is no previous number in the pattern prior to `2`, so no action is taken.
`plus3.currentNumber();`	`2`	The current number is `2`.

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

Copy of Semester1 Final QuestionBank-FRQ

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

- Assume that the classes listed in the Java Quick Reference 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 will not receive full credit.

The `Bus` class simulates the activity of a bus. A bus moves back and forth along a single route, making stops along the way. The stops on the route are numbered consecutively starting from 1 up to and including a number that is provided when the `Bus` object is created. You may assume that the number of stops will always be greater than 1.

The bus starts at the first stop and is initially heading toward the last stop. At each step of the simulation, the bus is at a particular stop and is heading toward either the first or last stop. When the bus reaches the first or last stop, it reverses direction. The following table contains a sample code execution sequence and the corresponding results.

Copy of Semester1 Final QuestionBank-FRQ

Statement or Expression	Value returned (blank if no value)	Comment
`Bus bus1 = new Bus(3);`		The route for `bus1` has three stops numbered 1–3.
`bus1.getCurrentStop();`	1	`bus1` is at stop 1 (first stop on the route).
`bus1.move();`		`bus1` moves to the next stop (2).
`bus1.getCurrentStop();`	2	`bus1` is at stop 2.
`bus1.move();`		`bus1` moves to the next stop (3).
`bus1.getCurrentStop();`	3	`bus1` is at stop 3.
`bus1.move();`		`bus1` moves to the next stop (2).
`bus1.getCurrentStop();`	2	`bus1` is at stop 2.
`bus1.move();`		`bus1` moves to the next stop (1).
`bus1.move();`		`bus1` moves to the next stop (2).
`bus1.getCurrentStop();`	2	`bus1` is at stop 2.
`bus1.getCurrentStop();`	2	`bus1` is still at stop 2.
`Bus bus2 = new Bus(5);`		The route for `bus2` has five stops numbered 1–5.
`bus1.getCurrentStop();`	2	`bus1` is still at stop 2.
`bus2.getCurrentStop();`	1	`bus2` is at stop 1 (first stop on the route).

Write the complete `Bus` class, including the constructor and any required instance variables and methods. Your implementation must meet all specifications and conform to the example.

Copy of Semester1 Final QuestionBank-FRQ

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

Assume that the classes listed in the Java Quick Reference 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 will not receive full credit.

This question involves computing factorials and using factorials to compute the number of possible ways that items can be selected from a group of choices. You will write two methods in the `Combinatorics` class that follows.

```
public class Combinatorics
{
     /** Precondition: n is between 1 and 12, inclusive.
       * Returns the factorial of n, as described in part (a).
       */
     public static int factorial(int n)
     { /* to be implemented in part (a) */ }

     /** Precondition: n and r are between 1 and 12, inclusive.
       * Determines the number of ways r items can be selected
       * from n choices and prints the result, as described in part (b).
       */
     public static void numCombinations(int n, int r)
     { /* to be implemented in part (b) */ }
}
```

(a) In mathematics, the factorial of a positive integer *n*, denoted as *n*! , is the product of all positive integers less than or equal to *n*.

The factorial of `n` can be computed using the following rules.

Case I: If `n` is `1`, then the factorial of `n` is `1`.
Case II: If `n` is greater than `1`, then the factorial of `n` is equal to `n` times the factorial of `(n - 1)`.

The `factorial` method returns the factorial of `n`, as determined by case I and case II. Write the `factorial` method below. You are encouraged to implement this method recursively.

```
/** Precondition: n is between 1 and 12, inclusive.
 * Returns the factorial of n, as described in part (a).
 */
public static int factorial(int n)
```

(b) A combination is a selection of items from a group of choices when the order that the items are selected does not

matter. For example, if there are four available choices (A, B, C, and D), there are six different ways that two items can be selected (A and B, A and C, A and D, B and C, B and D, C and D). The number of possible combinations of r items from a group of n choices can be calculated according to the following rules.

If r is greater than n, then the number of possible combinations is 0.
If r is not greater than n, then the number of possible combinations is equal to $\frac{n!}{r!(n-r)!}$.

The `numCombinations` method is intended to calculate the number of possible combinations of `r` items from a group of `n` choices and print the result. Examples of the intended behavior of the method are shown in the table.

Method Call	Message Printed	Explanation
`numCombinations(2, 4)`	`There are 0 ways of choosing 4 items from 2 choices.`	There is no way to select 4 items from 2 choices since 4 is greater than 2.
`numCombinations(5, 3)`	`There are 10 ways of choosing 3 items from 5 choices.`	The number of possible ways to select 2 items from 5 choices is $\frac{5!}{3!(5-3)!}$, or $\frac{5!}{(3!)(2!)}$, which evaluates to 10.

Write the `numCombinations` method below. Assume that `factorial` works as specified, regardless of what you wrote in part (a). You must use `factorial` appropriately to receive full credit.

```
/** Precondition: n and r are between 1 and 12, inclusive.
 * Determines the number of ways r items can be selected
 * from n choices and prints the result, as described in part (b).
 */
public static void numCombinations(int n, int r)
```

Copy of Semester1 Final QuestionBank-FRQ

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

Assume that the classes listed in the Java Quick Reference 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 will not receive full credit.

Some applications use a controlled vocabulary to describe, or tag, things. A controlled vocabulary is a limited set of keywords from which appropriate tags can be chosen.

The `Vocab` class, shown below, contains methods used to analyze words in terms of their presence in a controlled vocabulary. You will write two methods of the `Vocab` class.

```
public class Vocab
{
     /** The controlled vocabulary for a Vocab object. */
     private String[] theVocab = { /* contents not shown */ };

     /** Searches for a string in theVocab. Returns true if its String
   parameter str
      * is an exact match to an element in theVocab and returns false
    otherwise.
      */
     public boolean findWord(String str)
     {
         /* implementation not shown */
     }

     /** Counts how many strings in wordArray are not found in theVocab,
   as described in
      * part (a).
      */
     public int countNotInVocab(String[] wordArray)
     {
         /* to be implemented in part (a) */
     }

     /** Returns an array containing strings from wordArray not found in
   theVocab,
      * as described in part (b).
      */
     public String[] notInVocab(String[] wordArray)
     {
         /* to be implemented in part (b) */
     }
```

```
}
```

The `countNotInVocab` method returns an `int` that contains the number of words in its parameter `wordArray` that are not found in the instance variable `theVocab`.

A helper method, `findWord`, has been provided. The `findWord` method searches for an individual string in `theVocab`, returning `true` if an exact match between its `String` parameter and an element of `theVocab` is found, and returning `false` otherwise.

(a) Write the `countNotInVocab` method. Assume that there are no duplicates in `wordArray`. You must use `findWord` appropriately to receive full credit.

```
/** Counts how many strings in wordArray are not found in theVocab, as
described in
* part (a).
*/
public int countNotInVocab(String[] wordArray)
```

The `notInVocab` method returns an array of `String` objects that contains only elements of its parameter `wordArray` that are not found in `theVocab`. The array that is returned by `notInVocab` should have exactly one element for each word in `wordArray` that is not found in `theVocab`. Assume that there are no duplicates in `wordArray`.

The following example illustrates the behavior of the `notInVocab` method.

`theVocab`:

"time"	"food"	"dogs"	"cats"	"health"	"plants"	"sports"

`wordArray`:

"dogs"	"toys"	"sun"	"plants"	"time"

Array returned by `notInVocab`:

"toys"	"sun"

(b) Write the `notInVocab` method. Assume that there are no duplicates in `wordArray`. You must call `findWord` and `countNotInVocab` appropriately in order to receive full credit.

```
/** Returns an array containing strings from wordArray not found in
theVocab,
* as described in part (b).
```

Copy of Semester1 Final QuestionBank-FRQ

```
*/
public String[] notInVocab(String[] wordArray)
```

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

Assume that the classes listed in the Java Quick Reference 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 will not receive full credit.

This question uses two classes: an `Item` class that represents an item that has a name and value and an `ItemGrid` class that manages a two-dimensional array of items. A definition of the `Item` class is shown below.

```
public class Item
{
     private String name;
     private int value;

     public Item(String itemName, int itemValue)
     {
         name = itemName;
         value = itemValue;
     }

     public String getName()
     {
         return name;
     }

     public int getValue()
     {
         return value;
     }
}
```

The `ItemGrid` class below uses the two-dimensional array `grid` to represent a group of `Item` objects.

```
public class ItemGrid
{
     private Item[][] grid;

     // Constructor not shown

     /** Returns true if xPos is a valid row index and yPos is a valid
       * column index and returns false otherwise.
```

Copy of Semester1 Final QuestionBank-FRQ

```
    */
  public boolean isValid(int xPos, int yPos)
  { /* implementation not shown */ }

  /** Compares the item in row r and column c to the items to its
    * left and to its right. Returns the name of the item with
    * the greatest value, as described in part (a).
    * Precondition: r and c are valid indices
    */
  public String mostValuableNeighbor(int r, int c)
  { /* to be implemented in part (a) */ }

  /** Returns the average value of all the items in grid,
    * as described in part (b).
    */
  public double findAverage()
  { /* to be implemented in part (b) */ }
}
```

(a) Write the `mostValuableNeighbor` method, which compares the item in row `r` and column `c` to the items to its left and to its right. The method determines which of the three items has the greatest value and returns its name. If more than one of the values have a value that is greatest, then any of their names can be returned. If the item has no item to its left, it is compared only to the item to its right. If the item has no item to its right, it is compared only to the item to its left.

The helper method `isValid` has been provided. The `isValid` method returns `true` if `xPos` is a valid row index and `yPos` is a valid column index in the two-dimensional array `grid`, and returns `false` otherwise.

Assume that the `ItemGrid` object `ig` has been created such that the two-dimensional array `grid` contains the following item objects.

	0	1	2	3
0	"acorn" 7	"book" 10	"carrot" 8	"desk" 9
1	"egg" 5	"flag" 8	"globe" 8	"harp" 9
2	"island" 7	"jacket" 19	"kale" 8	"lunch" 16

The following table shows some examples of the behavior of the `mostValuableNeighbor` method.

Copy of Semester1 Final QuestionBank-FRQ

Method Call	Return Value	Explanation
`ig.mostValuableNeighbor(0, 2)`	`"book"`	The item at row `0`, column `2` (`"carrot"`) is compared with the items to its left and right (`"book"` and `"desk"`). Of the three items, `"book"` has the greatest value (`10`).
`ig.mostValuableNeighbor(1, 1)`	`"flag"` or `"globe"`	The item at row `1`, column `1` (`"flag"`) is compared with the items to its left and right (`"egg"` and `"globe"`). Of the three items, both `"flag"` and `"globe"` have the greatest value (`8`), so either can be returned.
`ig.mostValuableNeighbor(2, 0)`	`"jacket"`	The item at row `2`, column `0` (`"island"`) has no item to its left, so it is only compared with the item to its right (`"jacket"`). Of the two items, `"jacket"` has the greater value (`19`).
`ig.mostValuableNeighbor(2, 3)`	`"lunch"`	The item at row `2`, column `3` (`"lunch"`) has no item to its right, so it is only compared with the item to its left (`"kale"`). Of the two items, `"lunch"` has the greater value (`16`).

Complete the `mostValuableNeighbor` method below. You must use `isValid` appropriately to receive full credit. Assume that `grid` has been initialized with at least three rows and three columns, and contains no `null` elements.

```
/** Compares the item in row r and column c to the items to its
 * left and to its right. Returns the name of the item with
 * the greatest value, as described in part (a).
 * Precondition: r and c are valid indices
 */
public String mostValuableNeighbor(int r, int c)
```

(b) Write the `findAverage` method, which returns the average value of all items in `grid`. For example, for the `ItemGrid` object `ig` shown in part (a), the `findAverage` method should return `9.5`, which is the average value of the twelve items in the 2D array.

Complete the `findAverage` method below.

```
/** Returns the average value of all the items in grid,
 * as described in part (b).
 */
```

Copy of Semester1 Final QuestionBank-FRQ

```
public double findAverage()
```

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

- Assume that the classes listed in the Java Quick Reference 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 will not receive full credit.

This question involves performing arithmetic operations on two-dimensional (2D) arrays of integers. You will write two static methods, both of which are in a class named `MatrixOp` (not shown).

(a) Write the method `diagonalOp`, which returns the sum of the products of the corresponding entries on the main diagonals of two given square 2D arrays that have the same dimensions. The main diagonal goes from the top-left corner to the bottom-right corner in a square 2D array.

For example, assume that `mat1` and `mat2` are properly defined 2D arrays containing the values shown below. The main diagonals have been shaded in gray.

`mat1`

2	4	2
8	**5**	1
4	2	**4**

`mat2`

-1	8	9
6	**3**	5
5	1	**2**

After the call `int sum = MatrixOp.diagonalOp(mat1, mat2);` `sum` would contain `21`, as illustrated below.

```
sum = (2 * -1) + (5 * 3) + (4 * 2) = 21
```

Complete method `diagonalOp`.

```
/** Returns an integer, as described in part (a).
 * Precondition: matA and matB are 2D arrays that are both square,
 * have at least one row, and have the same dimensions.
```

Copy of Semester1 Final QuestionBank-FRQ

```
 */
public static int diagonalOp(int[][] matA, int[][] matB)
```

(b) Write the method `expandMatrix`, which returns an expanded version of a given 2D array. To expand a 2D array, a new 2D array must be created and filled with values such that each element of the original 2D array occurs a total of four times in the new 2D array, arranged as shown in the example below.

For example, assume that `mat3` is a properly defined 2D array containing the values shown below.

mat3

	0	1	2
0	−1	2	3
1	5	4	6

After the call `int[][] mat4 = MatrixOp.expandMatrix(mat3);` the array `mat4` would contain the values shown below.

Copy of Semester1 Final QuestionBank-FRQ

mat4

	0	1	2	3	4	5
0	−1	−1	2	2	3	3
1	−1	−1	2	2	3	3
2	5	5	4	4	6	6
3	5	5	4	4	6	6

Complete method `expandMatrix`.

```
/** Returns a 2D array, as described in part (b).
 * Precondition: matA is a 2D array with at least one row and
 * at least one column.
 */
public static int[][] expandMatrix(int[][] matA)
```

Copy of Semester1 Final QuestionBank-FRQ

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

Assume that the classes listed in the Java Quick Reference 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 will not receive full credit.

The `Meal` class and its subclass `DeluxeMeal` are used to represent meals at a restaurant.

All `Meal` objects have the following attributes and methods.

A `String` variable representing the name of the entree included in the meal
A `double` variable representing the cost, in dollars, of the meal
A `toString` method that indicates information about the meal

The following table shows the intended behavior of the `Meal` class.

Statement	Result
`Meal burger = new Meal("hamburger", 7.99);`	A new `Meal` object is created to represent a hamburger that costs $7.99.
`burger.toString();`	The string `"hamburger meal, $7.99"` is returned.

(a) Write the complete `Meal` class. Your implementation must meet all specifications and conform to the behavior shown in the table.

A deluxe meal, represented by a `DeluxeMeal` object, includes a side dish and a drink for an additional cost of $3. The `DeluxeMeal` class is a subclass of `Meal`. The `DeluxeMeal` class contains two additional attributes not found in `Meal`:

A `String` variable representing the name of the side dish included in the meal
A `String` variable representing the name of the drink included in the meal

The following table shows the intended behavior of the `DeluxeMeal` class.

Statement	Result
`DeluxeMeal burritoCombo = new DeluxeMeal("burrito", "chips", "lemonade", 7.49);`	A new `DeluxeMeal` object is created to represent a deluxe meal including a burrito (entree), chips (side dish), and lemonade (drink). The cost of the burrito alone is $7.49, so the cost of the meal is set to $10.49.
`burritoCombo.toString();`	The string `"deluxe burrito meal, $10.49"` is returned.

(b) Write the complete `DeluxeMeal` class. Your implementation must meet all specifications and conform to the behavior shown in the table.

Copy of Semester1 Final QuestionBank-FRQ

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

Notes:

- Assume that the classes listed in the Java Quick Reference 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 will not receive full credit.

An array of positive integer values has the mountain property if the elements are ordered such that successive values increase until a maximum value (the peak of the mountain) is reached and then the successive values decrease. The Mountain class declaration shown below contains methods that can be used to determine if an array has the mountain property. You will implement two methods in the Mountain class.

```
public class Mountain
{
  /** @param array an array of positive integer values
   *  @param stop the last index to check
   *          Precondition: 0 ≤ stop < array.length
   *  @return true if for each j such that 0 ≤ j < stop, array[j] < array[j + 1] ;
   *          false otherwise
   */
  public static boolean isIncreasing(int[] array, int stop)
  {  /*  implementation not shown  */  }

  /** @param array an array of positive integer values
   *  @param start the first index to check
   *         Precondition: 0 ≤ start < array.length - 1
   *  @return true if for each j such that start ≤ j < array.length - 1,
   *               array[j] > array[j + 1];
   *          false otherwise
   */
  public static boolean isDecreasing(int[] array, int start)
  {  /*  implementation not shown  */  }

  /** @param array an array of positive integer values
   *          Precondition: array.length > 0
   *  @return the index of the first peak (local maximum) in the array, if it exists;
   *          -1 otherwise
   */
  public static int getPeakIndex(int[] array)
  {  /*  to be implemented in part (a)  */  }

  /** @param array an array of positive integer values
   *          Precondition: array.length > 0
   *  @return true if array contains values ordered as a mountain;
   *          false otherwise
   */
  public static boolean isMountain(int[] array)
  {  /*  to be implemented in part (b)  */  }

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

}
```

(a) Write the Mountain method getPeakIndex. Method getPeakIndex returns the index of the first peak found in the parameter array, if one exists. A peak is defined as an element whose value is greater than the value of the

element immediately before it and is also greater than the value of the element immediately after it. Method getPeakIndex starts at the beginning of the array and returns the index of the first peak that is found or -1 if no peak is found.
For example, the following table illustrates the results of several calls to getPeakIndex.

arr	getPeakIndex(arr)
{11, 22, 33, 22, 11}	2
{11, 22, 11, 22, 11}	1
{11, 22, 33, 55, 77}	-1
{99, 33, 55, 77, 120}	-1
{99, 33, 55, 77, 55}	3
{33, 22, 11}	-1

Complete method getPeakIndex below.

```
/** @param array an array of positive integer values
 *         Precondition: array.length > 0
 *  @return the index of the first peak (local maximum) in the array, if it exists;
 *          -1 otherwise
 */
public static int getPeakIndex(int[] array)
```

(b) Write the Mountain method isMountain. Method isMountain returns true if the values in the parameter array are ordered as a mountain; otherwise, it returns false. The values in array
are ordered as a mountain if all three of the following conditions hold.

- There must be a peak.
- The array elements with an index smaller than the peak's index must appear in increasing order.
- The array elements with an index larger than the peak's index must appear in decreasing order.

For example, the following table illustrates the results of several calls to isMountain.

Copy of Semester1 Final QuestionBank-FRQ

arr	isMountain(arr)
{1, 2, 3, 2, 1}	true
{1, 2, 1, 2, 1}	false
{1, 2, 3, 1, 5}	false
{1, 4, 2, 1, 0}	true
{9, 3, 5, 7, 5}	false
{3, 2, 1}	false

In writing isMountain, assume that getPeakIndex works as specified, regardless of what you wrote in part (a). Complete method isMountain below.

```
/** @param array an array of positive integer values
 *          Precondition: array.length > 0
 *  @return true if array contains values ordered as a mountain;
 *          false otherwise
 */
public static boolean isMountain(int[] array)
```

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

- Assume that the classes listed in the Java Quick Reference 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 will not receive full credit.

This question involves generating a list of possible nicknames from a person's name. For the purposes of this question, a person's name contains a first (given) name and a last (family) name. You may assume that all punctuation has been removed so that names contain only letters and that all letters are in uppercase.

A partial declaration of the `NicknameGenerator` class is shown below. You will write two methods of the `NicknameGenerator` class.

```
public class NicknameGenerator
{
     /** The person's first name in all uppercase letters, initialized
       * by the constructor.
       */
     private String firstName;

     /** The person's last name in all uppercase letters, initialized
       * by the constructor.
       */
     private String lastName;

     // Constructor not shown

     /** Returns the number of vowels in lastName. */
     private int numVowels()
     { /* implementation not shown */ }

     /** Returns the index of the first vowel in lastName.
       * Returns -1 if there are no vowels in lastName.
       */
     private int indexOfFirstVowel()
     { /* implementation not shown */ }

     /** Returns a list of shortened last names, as described
       * in part (a).
       */
     public ArrayList<String> shortLastNames()
     { /* to be implemented in part (a) */ }
```

Copy of Semester1 Final QuestionBank-FRQ

```
    /** Returns a list of nicknames, as described in part (b). */
    public ArrayList<String> nicknames()
    { /* to be implemented in part (b) */ }
}
```

Write the method `shortLastNames`, which returns a list of the shortened forms of a last name, according to the rules below. Each shortened last name must appear exactly once in the list.

The following rules are used to produce the list of shortened last names.

- If the last name contains fewer than two vowels, the list contains the complete last name as its only element.
- If the last name contains two or more vowels, the first shortened last name in the list contains all characters from the original last name up to and including the first vowel. Each subsequent shortened last name is produced by adding one additional character from the original last name. The list ends with the complete last name.

The `NicknameGenerator` class provides two helper methods. The method `numVowels` returns the number of vowels in `lastName`. The method `indexOfFirstVowel` returns the index of the first vowel in `lastName` or `-1` if there are no vowels in `lastName`.

In the table below, the vowels in each last name are underlined.

Last name	List of shortened last names
NG	`["NG"]`
SMITH	`["SMITH"]`
LOPES	`["LO", "LOP", "LOPE", "LOPES"]`
ASHE	`["A", "AS", "ASH", "ASHE"]`

Complete method `shortLastNames`. The helper methods `numVowels` and `indexOfFirstVowel` have been provided for you.

```
/** Returns a list of shortened last names, as described
 * in part (a).
 */
public ArrayList<String> shortLastNames()
```

Write the method `nicknames`, which returns a list of all possible nicknames. Each nickname must appear exactly once in the list.

A nickname is generated with the first letter of the first name, followed by `"-"`, followed by one of the possible shortened forms of the last name. The table below shows several names and the list of nicknames that are generated from the name.

Copy of Semester1 Final QuestionBank-FRQ

Name	List of nicknames
CELESTE NG	`["C-NG"]`
GLEN SMITH	`["G-SMITH"]`
JUANITA LOPES	`["J-LO", "J-LOP", "J-LOPE", "J-LOPES"]`
MARY ASHE	`["M-A", "M-AS", "M-ASH", "M-ASHE"]`

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

Complete method `nicknames`.

```
/** Returns a list of nicknames, as described in part (b). */
public ArrayList<String> nicknames()
```

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

Assume that the classes listed in the Java Quick Reference 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 will not receive full credit.

This question involves the `StringManip` class, which is used to perform manipulation on strings.

The class provides the `removeSpaces` method, whose implementation is not shown. The method takes a string and returns a new string with spaces removed. For example, `removeSpaces("hi how are you")` returns `"hihowareyou"`. The `removeSpaces` method will be used in part (b).

```
public class StringManip
{
     /** Takes a string str and returns a new string
       * with all spaces removed.
       */
     public static String removeSpaces(String str)
     { /* implementation not shown */ }

     /** Takes a string str and returns a new string
       * with the characters reversed, as described in part (a).
       */
     public static String reverseString(String str)
     { /* to be implemented in part (a) */ }

     /** Determines whether str is a palindrome and prints a message
       * indicating the result, as described in part (b).
       * Precondition: str contains only lowercase letters and spaces.
       */
     public static void palindromeChecker(String str)
     { /* to be implemented in part (b) */ }
}
```

(a) Write method `reverseString`, which takes a string `str` and returns a new string with the characters in `str` in reverse order. For example, `reverseString("ABCDE")` should return `"EDCBA"`.

Complete the `reverseString` method below by assigning the reversed string to `result`.

```
/** Takes a string str and returns a new string
 * with the characters reversed.
 */
```

Copy of Semester1 Final QuestionBank-FRQ

```
public static String reverseString(String str)
{
     String result = "";
     return result;
}
```

For this question, let a *palindrome* be defined as a string that, when spaces are removed, reads the same forward and backward. For example, `"race car"` and `"taco cat"` are palindromes. You will write method `palindromeChecker`, which determines whether a string is a palindrome and prints a message indicating the result. Examples of the intended behavior of the method are shown in the following table.

Method Call	Printed Message
`palindromeChecker("taco cat")`	`taco cat is a palindrome`
`palindromeChecker("laid on no dial")`	`laid on no dial is a palindrome`
`palindromeChecker("level up")`	`level up is not a palindrome`

(b) Write method `palindromeChecker` below. Assume that `reverseString` works as specified, regardless of what you wrote in part (a). You must use `reverseString` and `removeSpaces` appropriately to receive full credit. Your implementation must conform to the examples in the table.

```
/** Determines whether str is a palindrome and prints a message
 * indicating the result, as described in part (b).
 * Precondition: str contains only lowercase letters and spaces.
 */
public static void palindromeChecker(String str)
```

Copy of Semester1 Final QuestionBank-FRQ

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

Assume that the classes listed in the Java Quick Reference 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 will not receive full credit.

This question involves the scheduling of car repairs. The classes used in the question are used to record information about car repairs. The methods shown and the methods to be written involve the mechanic performing a repair and the *bay* in which the repair takes place. A bay is an area of a repair shop in which a car is parked while a mechanic performs a repair. Mechanics and bays are identified by sequential integer identification numbers that start with `0`.

An individual car repair is represented by the following `CarRepair` class.

```
public class CarRepair
{
     private int mechanicNum;
     private int bayNum;

     public CarRepair(int m, int b)
     {
         mechanicNum = m;
         bayNum = b;
     }

     public int getMechanicNum()
     { return mechanicNum; }

     public int getBayNum()
     { return bayNum; }

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

The following `RepairSchedule` class represents the use of bays by mechanics repairing cars. You will write two methods of the `RepairSchedule` class.

```
public class RepairSchedule
{
     /** Each element represents a repair by an individual mechanic in a
  bay. */
     private ArrayList<CarRepair> schedule;
```

```
    /** Number of mechanics available in this schedule. */
    private int numberOfMechanics;

    /** Constructs a RepairSchedule object.
     * Precondition: n >= 0
     */
    public RepairSchedule(int n)
    {
        schedule = new ArrayList<CarRepair>();
        numberOfMechanics = n;
    }

    /** Attempts to schedule a repair by a given mechanic in a given bay
as described in part (a).
     * Precondition: 0 <= m < numberOfMechanics and b >= 0
     */
    public boolean addRepair(int m, int b)
    {
        /* to be implemented in part (a) */
    }

    /** Returns an ArrayList containing the mechanic identifiers of all
available mechanics,
     * as described in part (b).
     */
    public ArrayList<Integer> availableMechanics()
    {
        /* to be implemented in part (b) */
    }

    /** Removes an element from schedule when a repair is complete. */
    public void carOut(int b)
    {
        /* implementation not shown */
    }
}
```

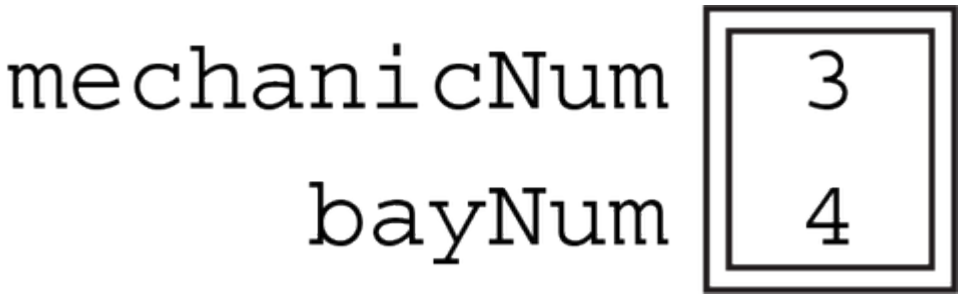
(a) Write the `addRepair` method. The method attempts to schedule a repair by the mechanic with identifier `m` in the bay with identifier `b`. The repair can be scheduled if mechanic `m` and bay `b` are both available. A mechanic is available if the given mechanic number does not appear in an element of `schedule` and a bay is available if the given bay number does not appear in an element of `schedule`.

If the mechanic and bay are both available, the `addRepair` method adds the repair to `schedule` and returns `true`. If either the mechanic or the bay are not available, the `addRepair` method returns `false`.

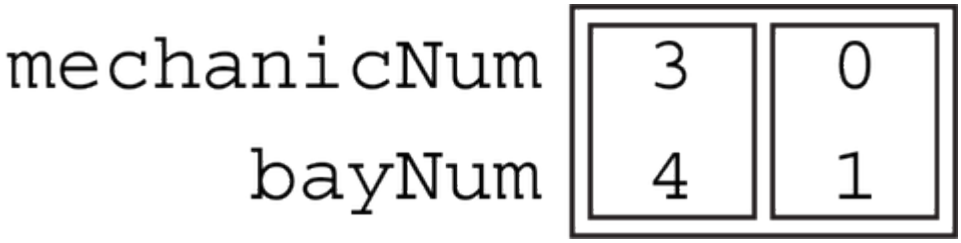
The following sequence of statements provides examples of the behavior of the `addRepair` method.

Copy of Semester1 Final QuestionBank-FRQ

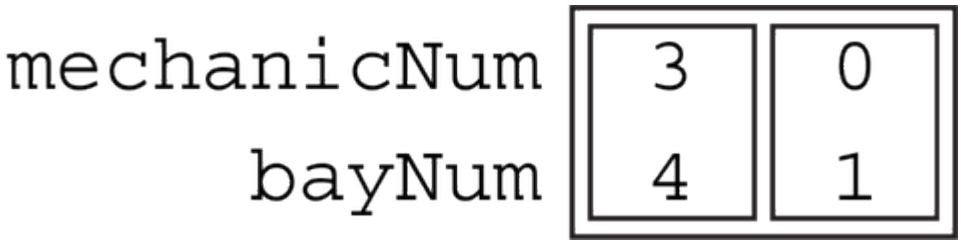
The statement `RepairSchedule r = new RepairSchedule(6);` constructs a new `RepairSchedule` object `r`. No repairs have been scheduled. Mechanics are numbered 0 through 5.
The call `r.addRepair(3, 4)` returns `true` because neither mechanic `3` nor bay `4` are present in `schedule`.
The contents of `schedule` after the call are as follows.



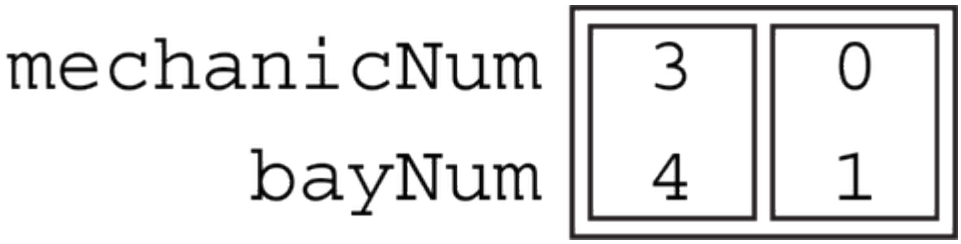
The call `r.addRepair(0, 1)` returns `true` because neither mechanic `0` nor bay `1` are present in `schedule`.
The contents of `schedule` after the call are as follows.



The call `r.addRepair(0, 2)` returns `false` because mechanic `0` is present in `schedule`. The contents of `schedule` after the call are as follows.

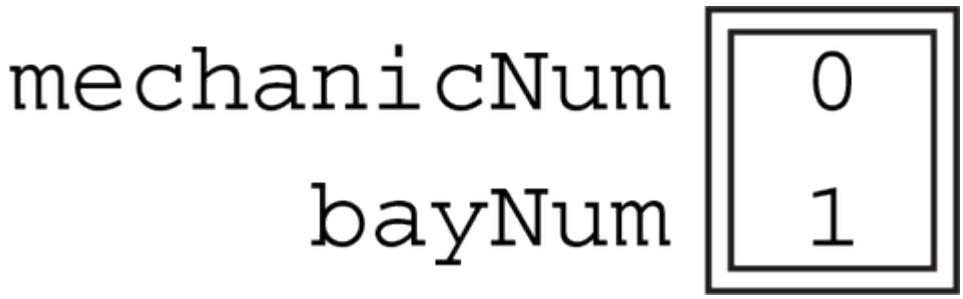


The call `r.addRepair(2, 4)` returns `false` because bay `4` is present in `schedule`. The contents of `schedule` after the call are as follows.

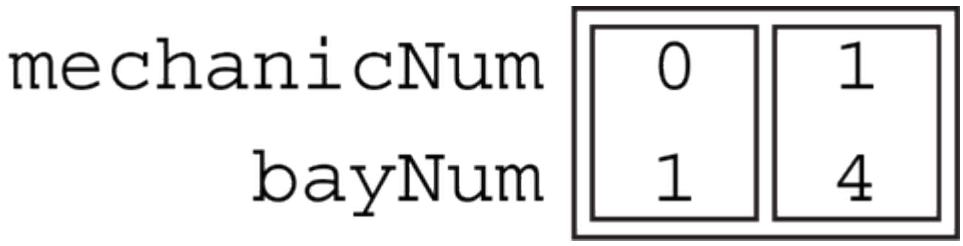


The call `r.carOut(4)` removes the repair in bay `4` from `schedule`. The `carOut` method is shown here to illustrate that bays and mechanics become available when car repairs are complete. You do not need to write or call this method. The contents of `schedule` after the call are as follows.

Copy of Semester1 Final QuestionBank-FRQ



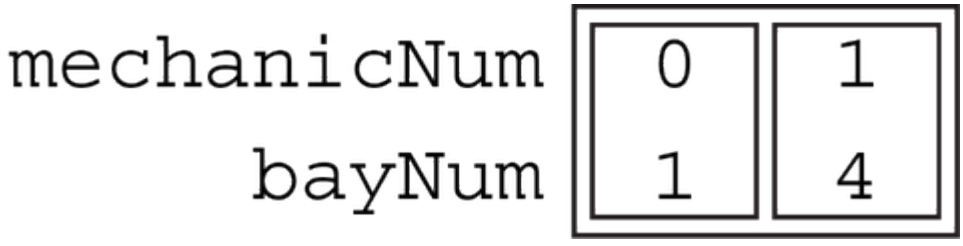
The call `r.addRepair(1, 4)` returns `true` because neither mechanic `1` nor bay `4` are present in `schedule`. The contents of `schedule` after the call are as follows.



Complete the `addRepair` method.

```
   /** Attempts to schedule a repair by a given mechanic in a given bay
as described in part (a).
     * Precondition: 0 <= m < numberOfMechanics and b >= 0
     */
   public boolean addRepair(int m, int b)
```

(b) Write the `availableMechanics` method, which returns an `ArrayList` containing the mechanic numbers of all available mechanics. If there is no available mechanic, an empty list is returned. A mechanic is available if the mechanic's identifier does not appear in `schedule`. Suppose `schedule` has the following contents.



For these contents of `schedule`, `availableMechanic` should return an `ArrayList` containing the values `2`, `3`, `4`, and `5` (in any order).

Complete the `availableMechanics` method. Assume that `addRepair` works as specified, regardless of what you wrote in part (a).

```
   /** Returns an ArrayList containing the mechanic identifiers of all
available mechanics,
```

Copy of Semester1 Final QuestionBank-FRQ

```
  * as described in part (b).
  */
public ArrayList<Integer> availableMechanics()
```

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

- Assume that the classes listed in the Java Quick Reference 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 will not receive full credit.

A manufacturer wants to keep track of the average of the ratings that have been submitted for an item using a running average. The algorithm for calculating a running average differs from the standard algorithm for calculating an average, as described in part (a).

A partial declaration of the `RunningAverage` class is shown below. You will write two methods of the `RunningAverage` class.

```
public class RunningAverage
{
     /** The number of ratings included in the running average. */
     private int count;

     /** The average of the ratings that have been entered. */
     private double average;

     // There are no other instance variables.

     /** Creates a RunningAverage object.
       * Postcondition: count is initialized to 0 and average is
       * initialized to 0.0.
       */
     public RunningAverage()
     { /* implementation not shown */ }

     /** Updates the running average to reflect the entry of a new
       * rating, as described in part (a).
       */
     public void updateAverage(double newVal)
     { /* to be implemented in part (a) */ }

     /** Processes num new ratings by considering them for inclusion
       * in the running average and updating the running average as
       * necessary. Returns an integer that represents the number of
       * invalid ratings, as described in part (b).
       * Precondition: num > 0
       */
```

```
    public int processNewRatings(int num)
    { /* to be implemented in part (b) */ }

    /** Returns a single numeric rating. */
    public double getNewRating()
    { /* implementation not shown */ }
}
```

(a) Write the method `updateAverage`, which updates the `RunningAverage` object to include a new rating. To update a running average, add the new rating to a calculated total, which is the number of ratings times the current running average. Divide the new total by the incremented count to obtain the new running average.

For example, if there are 4 ratings with a current running average of 3.5, the calculated total is 4 times 3.5, or 14.0. When a fifth rating with a value of 6.0 is included, the new total becomes 20.0. The new running average is 20.0 divided by 5, or 4.0.

Complete method `updateAverage`.

```
/** Updates the running average to reflect the entry of a new
 * rating, as described in part (a).
 */
public void updateAverage(double newVal)
```

(b) Write the `processNewRatings` method, which considers `num` new ratings for inclusion in the running average. A helper method, `getNewRating`, which returns a single rating, has been provided for you.

The running average must only be updated with ratings that are greater than or equal to zero. Ratings that are less than 0 are considered invalid and are not included in the running average.

The `processNewRatings` method returns the number of invalid ratings. See the table below for three examples of how calls to `processNewRatings` should work.

Copy of Semester1 Final QuestionBank-FRQ

Statement	Ratings Generated	processNewRatings Return Value	Comments
`processNewRatings(2)`	`2.5, 4.5`	`0`	Both new ratings are included in the running average.
`processNewRatings(1)`	`-2.0`	`1`	No new ratings are included in the running average.

`processNewRatings(4)`	`0.0,`
	`-2.2,`
	
	`3.5,`
	`-1.5`
		`2`	Two new ratings (`0.0` and `3.5`) are included in the running average.

Complete method `processNewRatings`. Assume that `updateAverage` works as specified, regardless of what you wrote in part (a). You must use `getNewRating` and `updateAverage` appropriately to receive full credit.

```
/** Processes num new ratings by considering them for inclusion
 * in the running average and updating the running average as
 * necessary. Returns an integer that represents the number of
 * invalid ratings, as described in part (b).
 * Precondition: num > 0
 */
public int processNewRatings(int num)
```

Copy of Semester1 Final QuestionBank-FRQ

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

Assume that the classes listed in the Java Quick Reference 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 will not receive full credit.

This question involves a game that is played with multiple spinners. You will write two methods in the `SpinnerGame` class below.

```
public class SpinnerGame
{
     /** Precondition: min < max
       * Simulates a spin of a spinner by returning a random integer
       * between min and max, inclusive.
       */
     public int spin(int min, int max)
     { /* to be implemented in part (a) */ }

     /** Simulates one round of the game as described in part (b).
       */
     public void playRound()
     { /* to be implemented in part (b) */ }
}
```

(a) The `spin` method simulates a spin of a fair spinner. The method returns a random integer between `min` and `max`, inclusive. Complete the `spin` method below by assigning this random integer to `result`.

```
/** Precondition: min < max
 * Simulates a spin of a spinner by returning a random integer
 * between min and max, inclusive.
 */
public int spin(int min, int max)
{
     int result;
  return result;
}
```

In each round of the game, the player and the computer each spin a spinner. The player spins a spinner numbered 1 to 10 , inclusive, whereas the computer spins a spinner numbered 2 to 8, inclusive.

Based on the results of the spins, a message is printed in the formats shown in the examples below.

If the player obtains a higher result than the computer, the player gains a number of points equal to the positive difference between the spins. If the computer obtains a higher result than the player, the player loses a number of points equal to the positive difference between the spins.

In the event of a tie, the player and the computer each spin the spinner a second time. If the sum of the player's two spins are greater than the sum of the computer's two spins, the player gains one point. If the sum of the computer's two spins are greater than the sum of the player's two spins, the player loses one point. In the event of a tie after two spins, the round is reported as a tie and the player's score does not change.

Examples of the `playRound` method's intended behavior are shown in the following table.

Player Spin #1	Computer Spin #1	Player Spin #2	Computer Spin #2	Printed String
9	6			`You win! 3 points`
3	7			`You lose. -4 points`
4	4	6	2	`You win! 1 points`
6	6	1	2	`You lose. -1 points`
1	1	8	8	`Tie. 0 points`

(b) Complete the `playRound` method below. You must use the `spin` method appropriately in order to earn full credit.

```
/** Simulates one round of the game as described in part (b).
 */
public void playRound()
```