FRQ (Unit 3) Boolean Expressions and if Statements

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.

Many encoded strings contain *delimiters*. A delimiter is a non-empty string that acts as a boundary between different parts of a larger string. The delimiters involved in this question occur in pairs that must be *balanced*, with each pair having an open delimiter and a close delimiter. There will be only one type of delimiter for each string. The following are examples of delimiters.

Example 1
Expressions in mathematics use open parentheses `"("` and close parentheses `")"` as delimiters. For each open parenthesis, there must be a matching close parenthesis.

`(x + y) * 5`	is a valid mathematical expression.
`(x + (y)`	is NOT a valid mathematical expression because there are more open delimiters than close delimiters.

Example 2
HTML uses `<B>` and `</B>` as delimiters. For each open delimiter `<B>`, there must be a matching close delimiter `</B>`.

`<B> Make this text bold </B>`	is valid HTML.
`<B> Make this text bold </UB>`	is NOT valid HTML because there is one open delimiter and no matching close delimiter.

In this question, you will write two methods in the following `Delimiters` class.

```
public class Delimiters
{

    /** The open and close delimiters. */
```

FRQ (Unit 3) Boolean Expressions and if Statements

```
    private String openDel;
    private String closeDel;

    /** Constructs a Delimiters object where open is the open delimiter
 and close is the
      *  close delimiter.
      *  Precondition: open and close are non-empty strings.
      */
    public Delimiters(String open, String close)
    {
        openDel = open;
        closeDel = close;
    }

    /** Returns an ArrayList of delimiters from the array tokens, as
 described in part (a). */
    public ArrayList<String> getDelimitersList(String[] tokens)
    {  /* to be implemented in part (a) */  }

    /** Returns true if the delimiters are balanced and false otherwise,
 as described in part (b).
      *  Precondition: delimiters contains only valid open and close
  delimiters.
      */
    public boolean isBalanced(ArrayList<String> delimiters)
    {  /* to be implemented in part (b) */  }

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

}
```

(a)

A string containing text and possibly delimiters has been split into *tokens* and stored in `String[] tokens`. Each token is either an open delimiter, a close delimiter, or a substring that is not a delimiter. You will write the method `getDelimitersList`, which returns an `ArrayList` containing all the open and close delimiters found in `tokens` in their original order.

The following examples show the contents of an `ArrayList` returned by `getDelimitersList` for different open and close delimiters and different `tokens` arrays.

Example 1

FRQ (Unit 3) Boolean Expressions and if Statements

openDel: `"("`

closeDel: `")"`

tokens:

"("	"x + y"	")"	" * 5"

ArrayList of delimiters:

"("	")"

Example 2

openDel: `"<q>"`

closeDel: `"</q>"`

tokens:

"<q>"	"yy"	"</q>"	"zz"	"</q>"

ArrayList of delimiters:

"<q>"	"</q>"	"</q>"

Class information for this question

```
public class Delimiters
private String openDel
private String closeDel
public Delimiters(String open, String close)
public ArrayList<String> getDelimitersList(String[] tokens)
public boolean isBalanced(ArrayList<String> delimiters)
```

Complete method `getDelimitersList` below.

```
/** Returns an ArrayList of delimiters from the array tokens, as
described in part (a). */
public ArrayList<String> getDelimitersList(String[] tokens)
```

(b)

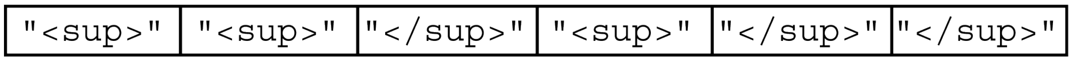
Write the method `isBalanced`, which returns `true` when the delimiters are balanced and returns `false` otherwise. The delimiters are balanced when both of the following conditions are satisfied; otherwise, they are not balanced.

1. When traversing the `ArrayList` from the first element to the last element, there is no point at which there are more close delimiters than open delimiters at or before that point.
2. The total number of open delimiters is equal to the total number of close delimiters.

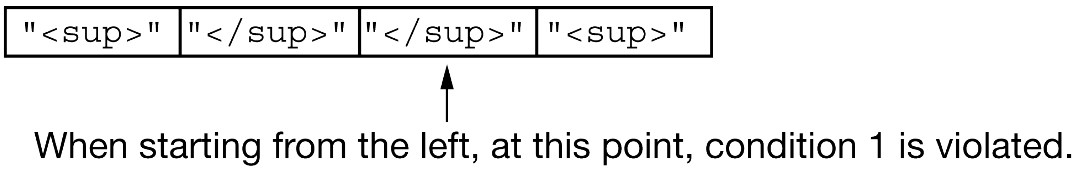
FRQ (Unit 3) Boolean Expressions and if Statements

Consider a `Delimiters` object for which `openDel` is "`<sup>`" and `closeDel` is "`</sup>`". The examples below show different `ArrayList` objects that could be returned by calls to `getDelimitersList` and the value that would be returned by a call to `isBalanced`.

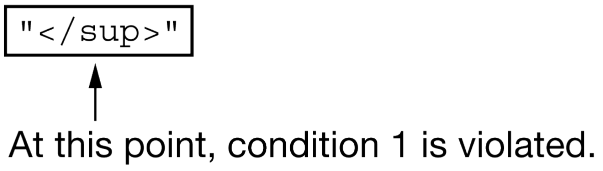
Example 1
The following example shows an `ArrayList` for which `isBalanced` returns `true`. As tokens are examined from first to last, the number of open delimiters is always greater than or equal to the number of close delimiters. After examining all tokens, there are an equal number of open and close delimiters.



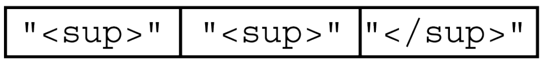
Example 2
The following example shows an `ArrayList` for which `isBalanced` returns `false`.


"<sup>"
"</sup>"
"</sup>"
"<sup>"
When starting from the left, at this point, condition 1 is violated.

Example 3
The following example shows an `ArrayList` for which `isBalanced` returns `false`.



Example 4
The following example shows an `ArrayList` for which `isBalanced` returns `false` because the second condition is violated. After examining all tokens, there are not an equal number of open and close delimiters.



Class information for this question

```
public class Delimiters
private String openDel
private String closeDel
public Delimiters(String open, String close)
```

FRQ (Unit 3) Boolean Expressions and if Statements

```
public ArrayList<String> getDelimitersList(String[] tokens)
public boolean isBalanced(ArrayList<String> delimiters)
```

Complete method `isBalanced` below.

```
/** Returns true if the delimiters are balanced and false otherwise, as
described in part (b).
 *  Precondition: delimiters contains only valid open and close
 delimiters.
 */
public boolean isBalanced(ArrayList<String> delimiters)
```

FRQ (Unit 3) Boolean Expressions and if Statements

2. **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.

FRQ (Unit 3) Boolean Expressions and if Statements

```
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.

FRQ (Unit 3) Boolean Expressions and if Statements

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.

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

FRQ (Unit 3) Boolean Expressions and if Statements

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)
```

FRQ (Unit 3) Boolean Expressions and if Statements

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

Notes:

- Assume that the classes listed in the appendices have been imported where appropriate.
- Unless otherwise noted in the question, assume that parameters in method calls are not null and that methods are called only when their preconditions are satisfied.
- In writing solutions for each question, you may use any of the accessible methods that are listed in classes defined in that question. Writing significant amounts of code that can be replaced by a call to one of these methods may not receive full credit.

Consider a software system that models a horse barn. Classes that represent horses implement the following interface.

```
public interface Horse
{
   /** @return the horse's name */
   String getName();

   /** @return the horse's weight */
   int getWeight();

   // There may be methods that are not shown.
}
```

A horse barn consists of *N* numbered spaces. Each space can hold at most one horse. The spaces are indexed starting from 0; the index of the last space is *N* – 1. No two horses in the barn have the same name.

The declaration of the HorseBarn class is shown below. You will write two unrelated methods of the HorseBarn class.

FRQ (Unit 3) Boolean Expressions and if Statements

```
public class HorseBarn
{
  /** The spaces in the barn. Each array element holds a reference to the horse
   *  that is currently occupying the space. A null value indicates an empty space.
   */
  private Horse[] spaces;

  /** Returns the index of the space that contains the horse with the specified name.
   *  Precondition: No two horses in the barn have the same name.
   *  @param name the name of the horse to find
   *  @return the index of the space containing the horse with the specified name;
   *             -1 if no horse with the specified name is in the barn.
   */
  public int findHorseSpace(String name)
  {  /* to be implemented in part (a) */  }

  /** Consolidates the barn by moving horses so that the horses are in adjacent spaces,
   *  starting at index 0, with no empty space between any two horses.
   *  Postcondition: The order of the horses is the same as before the consolidation.
   */
  public void consolidate()
  {  /* to be implemented in part (b) */  }

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

a. Write the HorseBarn method findHorseSpace. This method returns the index of the space in which the horse with the specified name is located. If there is no horse with the specified name in the barn, the method returns -1.

For example, assume a HorseBarn object called sweetHome has horses in the following spaces.

0	1	2	3	4	5	6
"Trigger" 1340	null	"Silver" 1210	"Lady" 1575	null	"Patches" 1350	"Duke" 1410

The following table shows the results of several calls to the findHorseSpace method.

FRQ (Unit 3) Boolean Expressions and if Statements

Method Call	Value Returned	Reason
`sweetHome.findHorseSpace("Trigger")`	0	A horse named Trigger is in space 0.
`sweetHome.findHorseSpace("Silver")`	2	A horse named Silver is in space 2.
`sweetHome.findHorseSpace("Coco")`	-1	A horse named Coco is not in the barn.

Information repeated from the beginning of the question

```
public interface Horse

String getName()
int getWeight()

public class HorseBarn

private Horse[] spaces
public int findHorseSpace(String name)
public void consolidate()
```

Complete method findHorseSpace below.

```
/** Returns the index of the space that contains the horse with the specified name.
 *  Precondition: No two horses in the barn have the same name.
 *  @param name  the name of the horse to find
 *  @return  the index of the space containing the horse with the specified name;
 *           -1  if no horse with the specified name is in the barn.
 */
public int findHorseSpace(String name)
```

b. Write the HorseBarn method consolidate. This method consolidates the barn by moving horses so that the horses are in adjacent spaces, starting at index 0, with no empty spaces between any two horses. After the barn is consolidated, the horses are in the same order as they were before the consolidation.

For example, assume a barn has horses in the following spaces.

0	1	2	3	4	5	6
"Trigger" 1340	null	"Silver" 1210	null	null	"Patches" 1350	"Duke" 1410

The following table shows the arrangement of the horses after consolidate is called.

FRQ (Unit 3) Boolean Expressions and if Statements

0	1	2	3	4	5	6
"Trigger" 1340	"Silver" 1210	"Patches" 1350	"Duke" 1410	null	null	null

Information repeated from the beginning of the question

```
public interface Horse

String getName()
int getWeight()

public class HorseBarn

private Horse[] spaces
public int findHorseSpace(String name)
public void consolidate()
```

Complete method consolidate below.

```
/** Consolidates the barn by moving horses so that the horses are in adjacent spaces,
 *  starting at index 0, with no empty space between any two horses.
 *  Postcondition: The order of the horses is the same as before the consolidation.
 */
public void consolidate()
```

FRQ (Unit 3) Boolean Expressions and if Statements

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 `Gizmo` class represents gadgets that people purchase. Some `Gizmo` objects are electronic and others are not. A partial definition of the `Gizmo` class is shown below.

```
public class Gizmo
{
     /** Returns the name of the manufacturer of this Gizmo. */
     public String getMaker()
     {
         /* implementation not shown */
     }

     /** Returns true if this Gizmo is electronic, and false otherwise. */
     public boolean isElectronic()
     {
         /* implementation not shown */
     }

     /** Returns true if this Gizmo is equivalent to the Gizmo object
  represented by the
      * parameter, and false otherwise.
      */
     public boolean equals(Object other)
     {
         /* implementation not shown */
     }

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

The `OnlinePurchaseManager` class manages a sequence of `Gizmo` objects that an individual has purchased from an online vendor. You will write two methods of the `OnlinePurchaseManager` class. A partial definition of the `OnlinePurchaseManager` class is shown below.

```
public class OnlinePurchaseManager
{
     /** An ArrayList of purchased Gizmo objects, instantiated in the
  constructor. */
     private ArrayList<Gizmo> purchases;

     /** Returns the number of purchased Gizmo objects that are electronic and
```

FRQ (Unit 3) Boolean Expressions and if Statements

```
     are
         * manufactured by maker, as described in part (a).
         */
       public int countElectronicsByMaker(String maker)
       {
           /* to be implemented in part (a) */
       }

       /** Returns true if any pair of adjacent purchased Gizmo objects are
     equivalent, and
         * false otherwise, as described in part (b).
         */
       public boolean hasAdjacentEqualPair()
       {
           /* to be implemented in part (b) */
       }

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

FRQ (Unit 3) Boolean Expressions and if Statements

4. **(a)** Write the `countElectronicsByMaker` method. The method examines the `ArrayList` instance variable `purchases` to determine how many `Gizmo` objects purchased are electronic and are manufactured by `maker`.

Assume that the `OnlinePurchaseManager` object `opm` has been declared and initialized so that the `ArrayList purchases` contains `Gizmo` objects as represented in the following table.

Index in `purchases`	**0**	**1**	**2**	**3**	**4**	**5**
Value returned by method call `isElectronic()`	`true`	`false`	`true`	`false`	`true`	`false`
Value returned by method call `getMaker()`	`"ABC"`	`"ABC"`	`"XYZ"`	`"lmnop"`	`"ABC"`	`"ABC"`

The following table shows the value returned by some calls to `countElectronicsByMaker`.

Method Call	**Return Value**
`opm.countElectronicsByMaker("ABC")`	2
`opm.countElectronicsByMaker("lmnop")`	0
`opm.countElectronicsByMaker("XYZ")`	1
`opm.countElectronicsByMaker("QRP")`	0

Complete method `countElectronicsByMaker` below.

```
/** Returns the number of purchased Gizmo objects that are electronic and
 * whose manufacturer is maker, as described in part (a).
 */
public int countElectronicsByMaker(String maker)
```

(b) When purchasing items online, users occasionally purchase two identical items in rapid succession without intending to do so (e.g., by clicking a purchase button twice). A vendor may want to check a user's purchase history to detect such occurrences and request confirmation.

Write the `hasAdjacentEqualPair` method. The method detects whether two adjacent `Gizmo` objects in `purchases` are equivalent, using the `equals` method of the `Gizmo` class. If an adjacent equivalent pair is found, the `hasAdjacentEqualPair` method returns `true`. If no such pair is found, or if `purchases` has fewer than two elements, the method returns `false`.

Complete method `hasAdjacentEqualPair` below.

```
/** Returns true if any pair of adjacent purchased Gizmo objects are
equivalent, and
 * false otherwise, as described in part (b).
 */
public boolean hasAdjacentEqualPair()
```