1. This question involves the `Account` class, which is used to represent user accounts for a website. The `Account` class contains a helper method, `isAvailable`, which determines whether a username is available. You will write a constructor and a method in the `Account` class.

```
public class Account
{
   private String username; // To be initialized in part (a)

   /**
    *  Creates a username based on the parameter requestedName. If the
    *  username is unavailable, appends successive integers, beginning
    *  with 1, to requestedName until an available username is found,
    *  as described in part (a).
    */
   public Account(String requestedName)
   { /* to be implemented in part (a) */ }

   /**
    *  Returns true if the parameter str is an available username;
    *  returns false otherwise.
    */
   public static boolean isAvailable(String str)
   { /* implementation not shown */ }

   /**
    *  Returns a shortened version of username with each hyphen ("-")
    *  and the character before it removed, as described in part (b)
    *  Preconditions: username does not start or end with a hyphen.
    *                 username does not contain consecutive hyphens.
    *                 username.length() >= 2
    *  Postcondition: username is unchanged.
    */
   public String getShortenedName()
   { /* to be implemented in part (b) */ }

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


The following information applies to part A.

In part A, you will write the `Account` constructor, which creates a username based on the parameter `requestedName`.

A helper method, `isAvailable`, has been provided to determine whether a username is available. The `isAvailable` method returns `true` if its parameter is an available username and returns `false` otherwise.

If `requestedName` is an available username, the `Account` constructor will assign `requestedName` to the instance variable `username`. If not, the constructor will try different variations of `requestedName` until an available username is found and assigned to the instance variable `username`. The variations consist of `requestedName` followed by an integer, as in the following example.

- If `requestedName` is `"Luis-Cruz"` and this username is not available, then the constructor will check the availability of `"Luis-Cruz1"`, `"Luis-Cruz2"`, `"Luis-Cruz3"`, and so on, until an available username is found. The first available username found is assigned to `username`.
- If `requestedName` is `"PSmith"` and this username is available, then `"PSmith"` is assigned to `username`.

The following information applies to part B.

In part B, you will write the `getShortenedName` method, which returns a shortened version of the instance variable `username` with each hyphen (`"-"`) and the character immediately before it removed. If no hyphens appear in `username`, the value of `username` is returned.

For example, if `username` is `"Amy-Marie-Lin"`, `getShortenedName` should return `"AmMariLin"`.

As another example, if `username` is `"SammyB3"`, `getShortenedName` should return `"SammyB3"`.

Part A

Complete the `Account` constructor. You must use `isAvailable` appropriately in order to receive full credit.

```
/**
 *  Creates a username based on the parameter requestedName. If the
 *  username is unavailable, appends successive integers, beginning
 *  with 1, to requestedName until an available username is found,
 *  as described in part (a).
 */
public Account(String requestedName)
```

Part B

Complete method `getShortenedName`.

```
/**
 *  Returns a shortened version of username with each hyphen ("-")
 *  and the character before it removed, as described in part (b)
 *  Preconditions: username does not start or end with a hyphen.
 *                 username does not contain consecutive hyphens.
 *                 username.length() >= 2
 *  Postcondition: username is unchanged.
 */
public String getShortenedName()
```


3. This question involves pairing competitors in a tournament into one-on-one matches for one round of the tournament. For example, in a chess tournament, the competitors are the individual chess players. A game of chess involving two players is a match. The winner of each match goes on to a match in the next round of the tournament. Since half of the players are eliminated in each round of the tournament, there is eventually a final round consisting of one match and two competitors. The winner of that match is considered the winner of the tournament.

Competitors, matches, and rounds of the tournament are represented by the `Competitor`, `Match`, and `Round` classes. You will write the constructor and one method of the `Round` class.

```
/** A single competitor in the tournament */
public class Competitor
{
  /** The competitor's name and rank */
  private String name;
  private int rank;

  /**
   * Assigns n to name and initialRank to rank
   * Precondition: initialRank >= 1
   */
  public Competitor(String n, int initialRank)
  { /* implementation not shown */ }

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



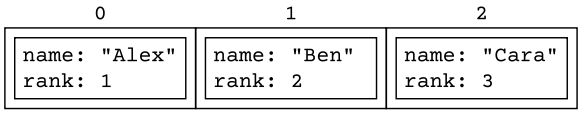
```
/** A match between two competitors */
public class Match
{
  public Match(Competitor one, Competitor two)
  { /* implementation not shown */ }

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


/** A single round of the tournament */
public class Round
{
  /** The list of competitors participating in this round */
  private ArrayList<Competitor> competitorList;

  /** Initializes competitorList, as described in part (a) */
  public Round(String[] names)
  { /* to be implemented in part (a) */ }

  /**
   * Creates an ArrayList of Match objects for the next round
   * of the tournament, as described in part (b)
   * Preconditions: competitorList contains at least one element.
   *                competitorList is ordered from best to worst rank.
   * Postcondition: competitorList is unchanged.
   */
  public ArrayList<Match> buildMatches()
  { /* to be implemented in part (b) */ }

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

A. Write the constructor for the `Round` class. The constructor should initialize `competitorList` to contain one `Competitor` object for each name in the `String[] names`. The order in which `Competitor` objects appear in `competitorList` should be the same as the order in which they appear in `names`, and the rank of each competitor is based on the competitor's position in `names`. Names are listed in `names` in order from the best-ranked competitor with rank `1` to the worst-ranked competitor with rank *n*, where *n* is the number of elements in `names`.

For example, assume the following code segment is executed.

```
String[] players = {"Alex", "Ben", "Cara"};
Round r = new Round(players);
```

The following shows the contents of `competitorList` in `r` after the constructor has finished executing.



Complete the `Round` constructor.

```
/** Initializes competitorList, as described in part (a) */
public Round(String[] names)
```

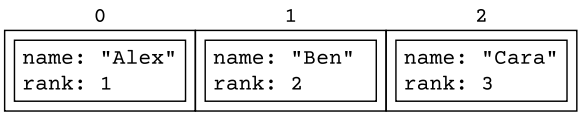

B. Write the `Round` method `buildMatches`. This method should return a new `ArrayList<Match>` object by pairing competitors from `competitorList` according to the following rules.

- If the number of competitors in `competitorList` is even, the best-ranked competitor is paired with the worst-ranked competitor, the second-best-ranked competitor is paired with the second-worst-ranked competitor, etc.
- If the number of competitors in `competitorList` is odd, the competitor with the best rank is ignored and the remaining competitors are paired according to the rule for an even number of competitors.

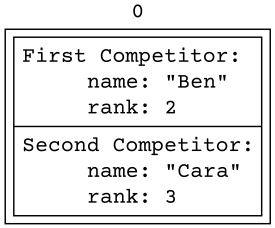
Each pair of competitors is used to create a `Match` object that should be added to the `ArrayList` to return. Competitors may appear in either order in a `Match` object, and matches may appear in any order in the returned `ArrayList`.

The following example shows the contents of `competitorList` in a `Round` object `r1` containing an odd number of competitors and the `ArrayList` of `Match` objects that should be returned by the call `r1.buildMatches()`.

`competitorList`:

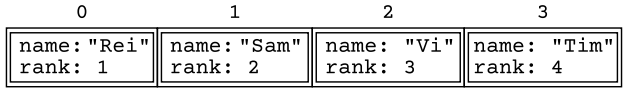


The `ArrayList` to be returned contains a single match between Ben and Cara, the second and third-ranked competitors:

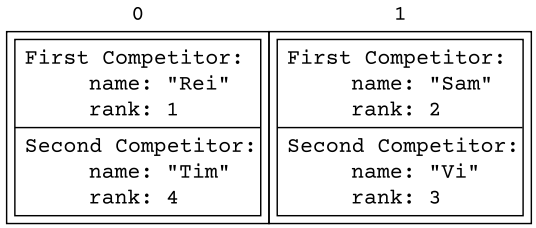


The next example shows the contents of `competitorList` in a `Round` object `r2` containing an even number of competitors and the `ArrayList` of `Match` objects that should be returned by the call `r2.buildMatches()`.

`competitorList`:



The `ArrayList` to be returned contains two matches: one match between the first- and last-ranked competitors Rei and Tim, and a second match between the second- and third-ranked competitors Sam and Vi:



Complete the `buildMatches` method.

```
/**
 * Creates an ArrayList of Match objects for the next round
 * of the tournament, as described in part (b)
 * Preconditions: competitorList contains at least one element.
 *                competitorList is ordered from best to worst rank.
 * Postcondition: competitorList is unchanged.
 */
public ArrayList<Match> buildMatches()
```


2. This question involves a scoreboard for a game. The game is played between two teams who alternate turns so that at any given time, one team is active and the other team is inactive. During a turn, a team makes one or more plays. Each play can score one or more points and the team's turn continues, or the play can fail, in which case no points are scored and the team's turn ends. The `Scoreboard` class, which you will write, is used to keep track of the score in a game.

   The `Scoreboard` class contains a constructor and two methods.

   - The constructor has two parameters. The first parameter is a `String` containing the name of team 1, and the second parameter is a `String` containing the name of team 2. The game always begins with team 1 as the active team.
   - The `recordPlay` method has a single nonnegative integer parameter that is equal to the number of points scored on a play or `0` if the play failed. If the play results in one or more points scored, the active team's score is updated and that team remains active. If the value of the parameter is `0`, the active team's turn ends and the inactive team becomes the active team. The `recordPlay` method does not return a value.
   - The `getScore` method has no parameters. The method returns a `String` containing information about the current state of the game. The returned string begins with the score of team 1, followed by a hyphen (`"-"`), followed by the score of team 2, followed by a hyphen, followed by the name of the team that is currently active.

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

Statement	Value Returned (blank if none)	Explanation
`String info;`		
`Scoreboard game = new Scoreboard("Red", "Blue");`		`game` is a new `Scoreboard` for a game played between team 1, whose name is `"Red"`, and team 2, whose name is `"Blue"`. The active team is set to team 1.
`info = game.getScore();`	`"0-0-Red"`	
`game.recordPlay(1);`		Team 1 earns 1 point because the game always begins with team 1 as the active team.
`info = game.getScore();`	`"1-0-Red"`	
`game.recordPlay(0);`		Team 1's play failed, so team 2 is now active.
`info = game.getScore();`	`"1-0-Blue"`	
`info = game.getScore();`	`"1-0-Blue"`	The score and state of the game are unchanged since the last call to `getScore`.
`game.recordPlay(3);`		Team 2 earns 3 points.
`info = game.getScore();`	`"1-3-Blue"`	
`game.recordPlay(1);`		Team 2 earns 1 point.
`game.recordPlay(0);`		Team 2's play failed, so team 1 is now active.
`info = game.getScore();`	`"1-4-Red"`	
`game.recordPlay(0);`		Team 1's play failed, so team 2 is now active.
`game.recordPlay(4);`		Team 2 earns 4 points.
`game.recordPlay(0);`		Team 2's play failed, so team 1 is now active.
`info = game.getScore();`	`"1-8-Red"`	
`Scoreboard match = new Scoreboard("Lions", "Tigers");`		`match` is a new and independent `Scoreboard` object.
`info = match.getScore();`	`"0-0-Lions"`	
`info = game.getScore();`	`"1-8-Red"`	

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

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

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

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

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

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

Width of the Sign	Sign Display

15	`Everything on s`
	`ale, please com`
	`e in`

17	`Everything on sal` `e, please come in`
40	`Everything on sale, please come in`

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

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

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

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

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

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

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

2. The `Book` class is used to store information about a book. A partial `Book` class definition is shown.

```
public class Book
{
   /** The title of the book */
   private String title;

   /** The price of the book */
   private double price;

   /** Creates a new Book with given title and price */
   public Book(String bookTitle, double bookPrice)
   {  /* implementation not shown */  }

   /** Returns the title of the book */
   public String getTitle()
   {  return title;  }

   /** Returns a string containing the title and price of the Book */
   public String getBookInfo()
   {
      return title + "-" + price;
   }

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

You will write a class `Textbook`, which is a subclass of `Book`.

A `Textbook` has an edition number, which is a positive integer used to identify different versions of the book. The `getBookInfo` method, when called on a `Textbook`, returns a string that also includes the edition information, as shown in the example.

Information about the book title and price must be maintained in the `Book` class. Information about the edition must be maintained in the `Textbook` class.

The `Textbook` class contains an additional method, `canSubstituteFor`, which returns `true` if a `Textbook` is a valid substitute for another `Textbook` and returns `false` otherwise. The current `Textbook` is a valid substitute for the `Textbook` referenced by the parameter of the `canSubstituteFor` method if the two `Textbook` objects have the same title and if the edition of the current `Textbook` is greater than or equal to the edition of the parameter.

GO ON TO THE NEXT PAGE.

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

Statement	Value Returned (blank if no value)	Class Specification
`Textbook bio2015 = new Textbook("Biology", 49.75, 2);`		`bio2015` is a `Textbook` with a title of `"Biology"`, a price of `49.75`, and an edition of `2`.
`Textbook bio2019 = new Textbook("Biology", 39.75, 3);`		`bio2019` is a `Textbook` with a title of `"Biology"`, a price of `39.75`, and an edition of `3`.
`bio2019.getEdition();`	`3`	The edition is returned.
`bio2019.getBookInfo();`	`"Biology-39.75-3"`	The formatted string containing the title, price, and edition of `bio2019` is returned.
`bio2019.canSubstituteFor(bio2015);`	`true`	`bio2019` is a valid substitute for `bio2015`, since their titles are the same and the edition of `bio2019` is greater than or equal to the edition of `bio2015`.
`bio2015.canSubstituteFor(bio2019);`	`false`	`bio2015` is not a valid substitute for `bio2019`, since the edition of `bio2015` is less than the edition of `bio2019`.
`Textbook math = new Textbook("Calculus", 45.25, 1);`		`math` is a `Textbook` with a title of `"Calculus"`, a price of `45.25`, and an edition of `1`.
`bio2015.canSubstituteFor(math);`	`false`	`bio2015` is not a valid substitute for `math`, since the title of `bio2015` is not the same as the title of `math`.

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

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

GO ON TO THE NEXT PAGE.

2. The class `SingleTable` represents a table at a restaurant.

```
public class SingleTable
{
    /** Returns the number of seats at this table. The value is always greater than or equal to 4. */
    public int getNumSeats()
    {  /* implementation not shown */  }

    /** Returns the height of this table in centimeters. */
    public int getHeight()
    {  /* implementation not shown */  }

    /** Returns the quality of the view from this table. */
    public double getViewQuality()
    {  /* implementation not shown */  }

    /** Sets the quality of the view from this table to value. */
    public void setViewQuality(double value)
    {  /* implementation not shown */  }

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

At the restaurant, customers can sit at tables that are composed of two single tables pushed together. You will write a class `CombinedTable` to represent the result of combining two `SingleTable` objects, based on the following rules and the examples in the chart that follows.

- A `CombinedTable` can seat a number of customers that is two fewer than the total number of seats in its two `SingleTable` objects (to account for seats lost when the tables are pushed together).
- A `CombinedTable` has a desirability that depends on the views and heights of the two single tables. If the two single tables of a `CombinedTable` object are the same height, the desirability of the `CombinedTable` object is the average of the view qualities of the two single tables.
- If the two single tables of a `CombinedTable` object are not the same height, the desirability of the `CombinedTable` object is 10 units less than the average of the view qualities of the two single tables.

Assume `SingleTable` objects `t1`, `t2`, and `t3` have been created as follows.

- `SingleTable t1` has 4 seats, a view quality of 60.0, and a height of 74 centimeters.
- `SingleTable t2` has 8 seats, a view quality of 70.0, and a height of 74 centimeters.
- `SingleTable t3` has 12 seats, a view quality of 75.0, and a height of 76 centimeters.

The chart contains a sample code execution sequence and the corresponding results.

Statement	Value Returned (blank if no value)	Class Specification
`CombinedTable c1 = new CombinedTable(t1, t2);`		A `CombinedTable` is composed of two `SingleTable` objects.
`c1.canSeat(9);`	`true`	Since its two single tables have a total of 12 seats, `c1` can seat 10 or fewer people.
`c1.canSeat(11);`	`false`	`c1` cannot seat 11 people.
`c1.getDesirability();`	`65.0`	Because `c1`'s two single tables are the same height, its desirability is the average of `60.0` and `70.0`.
`CombinedTable c2 = new CombinedTable(t2, t3);`		A `CombinedTable` is composed of two `SingleTable` objects.
`c2.canSeat(18);`	`true`	Since its two single tables have a total of 20 seats, `c2` can seat 18 or fewer people.
`c2.getDesirability();`	`62.5`	Because `c2`'s two single tables are not the same height, its desirability is 10 units less than the average of `70.0` and `75.0`.
`t2.setViewQuality(80);`		Changing the view quality of one of the tables that makes up `c2` changes the desirability of `c2`, as illustrated in the next line of the chart. Since `setViewQuality` is a `SingleTable` method, you do not need to write it.
`c2.getDesirability();`	`67.5`	Because the view quality of `t2` changed, the desirability of `c2` has also changed.

The last line of the chart illustrates that when the characteristics of a `SingleTable` change, so do those of the `CombinedTable` that contains it.

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

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

2. This question involves the implementation of a fitness tracking system that is represented by the `StepTracker` class. A `StepTracker` object is created with a parameter that defines the minimum number of steps that must be taken for a day to be considered *active*.

   The `StepTracker` class provides a constructor and the following methods.

   - `addDailySteps`, which accumulates information about steps, in readings taken once per day
   - `activeDays`, which returns the number of active days
   - `averageSteps`, which returns the average number of steps per day, calculated by dividing the total number of steps taken by the number of days tracked

   The following table contains a sample code execution sequence and the corresponding results.

Statements and Expressions	Value Returned (blank if no value)	Comment
`StepTracker tr = new StepTracker(10000);`		Days with at least 10,000 steps are considered active. Assume that the parameter is positive.
`tr.activeDays();`	`0`	No data have been recorded yet.
`tr.averageSteps();`	`0.0`	When no step data have been recorded, the `averageSteps` method returns `0.0`.
`tr.addDailySteps(9000);`		This is too few steps for the day to be considered active.
`tr.addDailySteps(5000);`		This is too few steps for the day to be considered active.
`tr.activeDays();`	`0`	No day had at least 10,000 steps.
`tr.averageSteps();`	`7000.0`	The average number of steps per day is (14000 / 2).
`tr.addDailySteps(13000);`		This represents an active day.
`tr.activeDays();`	`1`	Of the three days for which step data were entered, one day had at least 10,000 steps.
`tr.averageSteps();`	`9000.0`	The average number of steps per day is (27000 / 3).
`tr.addDailySteps(23000);`		This represents an active day.
`tr.addDailySteps(1111);`		This is too few steps for the day to be considered active.
`tr.activeDays();`	`2`	Of the five days for which step data were entered, two days had at least 10,000 steps.
`tr.averageSteps();`	`10222.2`	The average number of steps per day is (51111 / 5).

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

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

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

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

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

Example 1

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

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

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

Example 2

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

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

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

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

1. This question involves identifying and processing the digits of a non-negative integer. The declaration of the `Digits` class is shown below. You will write the constructor and one method for the `Digits` class.

```
public class Digits
{
    /** The list of digits from the number used to construct this object.
     *  The digits appear in the list in the same order in which they appear in the original number.
     */
    private ArrayList<Integer> digitList;

    /** Constructs a Digits object that represents num.
     *  Precondition: num >= 0
     */
    public Digits(int num)
    {  /* to be implemented in part (a) */  }

    /** Returns true if the digits in this Digits object are in strictly increasing order;
     *          false otherwise.
     */
    public boolean isStrictlyIncreasing()
    {  /* to be implemented in part (b) */  }
}
```

Part (a) begins on page 4.

(a) Write the constructor for the `Digits` class. The constructor initializes and fills `digitList` with the digits from the non-negative integer `num`. The elements in `digitList` must be `Integer` objects representing single digits, and appear in the same order as the digits in `num`. Each of the following examples shows the declaration of a `Digits` object and the contents of `digitList` as initialized by the constructor.

Example 1

```
Digits d1 = new Digits(15704);
```

`d1`:

	0	1	2	3	4
digitList:	1	5	7	0	4

Example 2

```
Digits d2 = new Digits(0);
```

`d2`:

	0
digitList:	0

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Complete the `Digits` constructor below.

```
/** Constructs a Digits object that represents num.
 *  Precondition: num >= 0
 */
public Digits(int num)
```

Part (b) begins on page 6.

(b) Write the `Digits` method `isStrictlyIncreasing`. The method returns `true` if the elements of `digitList` appear in strictly increasing order; otherwise, it returns `false`. A list is considered strictly increasing if each element after the first is greater than (but not equal to) the preceding element.

The following table shows the results of several calls to `isStrictlyIncreasing`.

Method call	Value returned
`new Digits(7).isStrictlyIncreasing()`	`true`
`new Digits(1356).isStrictlyIncreasing()`	`true`
`new Digits(1336).isStrictlyIncreasing()`	`false`
`new Digits(1536).isStrictlyIncreasing()`	`false`
`new Digits(65310).isStrictlyIncreasing()`	`false`

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Complete method `isStrictlyIncreasing` below.

```
/** Returns true if the digits in this Digits object are in strictly increasing order;
 *          false otherwise.
 */
```

2. This question involves the design of a class that will be used to produce practice problems. The following `StudyPractice` interface represents practice problems that can be used to study some subject.

```
public interface StudyPractice
{
    /** Returns the current practice problem. */
    String getProblem();

    /** Changes to the next practice problem. */
    void nextProblem();
}
```

The `MultPractice` class is a `StudyPractice` that produces multiplication practice problems. A `MultPractice` object is constructed with two integer values: *first integer* and *initial second integer*. The first integer is a value that remains constant and is used as the first integer in every practice problem. The initial second integer is used as the starting value for the second integer in the practice problems. This second value is incremented for each additional practice problem that is produced by the class.

For example, a `MultPractice` object created with the call `new MultPractice(7, 3)` would be used to create the practice problems `"7 TIMES 3"`, `"7 TIMES 4"`, `"7 TIMES 5"`, and so on.

In the `MultPractice` class, the `getProblem` method returns a string in the format of "*first integer* `TIMES` *second integer*". The `nextProblem` method updates the state of the `MultPractice` object to represent the next practice problem.

The following examples illustrate the behavior of the `MultPractice` class. Each table shows a code segment and the output that would be produced as the code is executed.

Example 1

Code segment	Output produced
`StudyPractice p1 = new MultPractice(7, 3);`	
`System.out.println(p1.getProblem());`	`7 TIMES 3`
`p1.nextProblem();`	
`System.out.println(p1.getProblem());`	`7 TIMES 4`
`p1.nextProblem();`	
`System.out.println(p1.getProblem());`	`7 TIMES 5`
`p1.nextProblem();`	
`System.out.println(p1.getProblem());`	`7 TIMES 6`

Example 2

Code segment	Output produced
`StudyPractice p2 = new MultPractice(4, 12);`	
`p2.nextProblem();`	
`System.out.println(p2.getProblem());`	`4 TIMES 13`
`System.out.println(p2.getProblem());`	`4 TIMES 13`
`p2.nextProblem();`	
`p2.nextProblem();`	
`System.out.println(p2.getProblem());`	`4 TIMES 15`
`p2.nextProblem();`	
`System.out.println(p2.getProblem());`	`4 TIMES 16`

Question 2 continues on page 10.

Interface information for this question

```
public interface StudyPractice

String getProblem()
void nextProblem()
```

Write the complete `MultPractice` class. Your implementation must be consistent with the specifications and the given examples.

COMPUTER SCIENCE A

SECTION II

Time—1 hour and 30 minutes

Number of questions—4

Percent of total score—50

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

Notes:

- Assume that the classes listed in the 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.

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 YOUR SOLUTION ON THE NEXT PAGE.

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.

Part (b) begins on page 4.

(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)
```

Part (b) begins on page 8.

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

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

- 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"`

WRITE YOUR SOLUTION ON THE NEXT PAGE.

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

Complete method `containsWord` below.

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

Part (c) begins on page 10.

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

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

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

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

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

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

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

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

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

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

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

Complete method `removeMessages` below.

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