

2. The `Bottle` class, which you will write, represents a bottle that contains liquid.

`Bottle` objects are created by calls to a constructor with a `double` parameter that represents the bottle's capacity, in milliliters (mL), which is the maximum amount of liquid in the bottle. Assume that this value will be greater than or equal to 0. When `Bottle` objects are constructed, each is filled to its capacity.

The `Bottle` class contains an `updateAmount` method, which updates the amount of liquid in the bottle. This method has a `double` parameter that represents the amount of liquid, in mL, to be removed from the bottle, with a precondition that the parameter is always greater than zero and less than or equal to the amount of liquid remaining in the bottle. If updating the amount of liquid in the bottle would cause it to be less than 25% of the capacity, the bottle is filled and reset to the capacity. The `updateAmount` method returns a `double` that represents the amount remaining in the bottle, in mL, after the amount has been updated.

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

Statement	Return Value (blank if no value)	Explanation
<code>double amt;</code>		
<code>Bottle water = new Bottle(1000.0);</code>		A <code>Bottle</code> object named <code>water</code> is constructed with a capacity of 1,000 mL of liquid.
<code>amt = water.updateAmount(400.0);</code>	600.0	400 mL are used, so 600 mL remain in the bottle.
<code>amt = water.updateAmount(100.0);</code>	500.0	100 mL are used, so 500 mL remain in the bottle.
<code>amt = water.updateAmount(300.0);</code>	1000.0	300 mL are used, so 200 mL remain in the bottle. Since this amount is less than 250 mL (25% of the capacity of 1,000 mL), the bottle is immediately filled and reset to the initial amount of 1,000 mL.
<code>Bottle shampoo = new Bottle(40.0);</code>		A <code>Bottle</code> object named <code>shampoo</code> is constructed with a capacity of 40 mL of liquid.
<code>amt = shampoo.updateAmount(30.0);</code>	10.0	30 mL are used, so 10 mL remain in the bottle.
<code>amt = shampoo.updateAmount(1.0);</code>	40.0	1 mL is used, so 9 mL remain in the bottle. Since this amount is less than 10 mL (25% of the capacity of 40 mL), the bottle is immediately filled and reset to the initial amount of 40 mL.

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

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

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

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

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

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

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

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

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

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

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

2. This question involves 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
<code>String info;</code> <code>Scoreboard game = new Scoreboard("Red", "Blue");</code>		game is a new <code>Scoreboard</code> 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.
<code>info = game.getScore();</code> <code>game.recordPlay(1);</code>	"0-0-Red"	
		Team 1 earns 1 point because the game always begins with team 1 as the active team.
<code>info = game.getScore();</code> <code>game.recordPlay(0);</code>	"1-0-Red"	
		Team 1's play failed, so team 2 is now active.
<code>info = game.getScore();</code> <code>info = game.getScore();</code>	"1-0-Blue"	
		The score and state of the game are unchanged since the last call to <code>getScore</code> .
<code>game.recordPlay(3);</code> <code>info = game.getScore();</code>	"1-3-Blue"	
		Team 2 earns 3 points.
<code>game.recordPlay(1);</code> <code>game.recordPlay(0);</code>		Team 2 earns 1 point.
		Team 2's play failed, so team 1 is now active.
<code>info = game.getScore();</code> <code>game.recordPlay(0);</code>	"1-4-Red"	
		Team 1's play failed, so team 2 is now active.
<code>game.recordPlay(4);</code> <code>game.recordPlay(0);</code>		Team 2 earns 4 points.
		Team 2's play failed, so team 1 is now active.
<code>info = game.getScore();</code> <code>Scoreboard match = new Scoreboard("Lions", "Tigers");</code>	"1-8-Red"	
		match is a new and independent <code>Scoreboard</code> object.
<code>info = match.getScore();</code> <code>info = game.getScore();</code>	"0-0-Lions"	
	"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 sale, please come in
17	Everything on sale, 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 sale, please come in". No semicolon should appear at the end of the `String` returned by `getLines`.

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

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

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

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

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
<code>StepTracker tr = new StepTracker(10000);</code>		Days with at least 10,000 steps are considered active. Assume that the parameter is positive.
<code>tr.activeDays();</code>	0	No data have been recorded yet.
<code>tr.averageSteps();</code>	0.0	When no step data have been recorded, the <code>averageSteps</code> method returns 0.0.
<code>tr.addDailySteps(9000);</code>		This is too few steps for the day to be considered active.
<code>tr.addDailySteps(5000);</code>		This is too few steps for the day to be considered active.
<code>tr.activeDays();</code>	0	No day had at least 10,000 steps.
<code>tr.averageSteps();</code>	7000.0	The average number of steps per day is (14000 / 2).
<code>tr.addDailySteps(13000);</code>		This represents an active day.
<code>tr.activeDays();</code>	1	Of the three days for which step data were entered, one day had at least 10,000 steps.
<code>tr.averageSteps();</code>	9000.0	The average number of steps per day is (27000 / 3).
<code>tr.addDailySteps(23000);</code>		This represents an active day.
<code>tr.addDailySteps(1111);</code>		This is too few steps for the day to be considered active.
<code>tr.activeDays();</code>	2	Of the five days for which step data were entered, two days had at least 10,000 steps.
<code>tr.averageSteps();</code>	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
<code>sc1.isValid("happy")</code>	true	The code word is valid.
<code>sc1.isValid("happy\$")</code>	false	The code word contains "\$".
<code>sc1.isValid("Code")</code>	false	The code word is too short.
<code>sc1.isValid("happyCode")</code>	false	The code word is too long.

Example 2

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

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

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

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

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 `MultiPractice` class is a `StudyPractice` that produces multiplication practice problems. A `MultiPractice` 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 `MultiPractice` object created with the call `new MultiPractice(7, 3)` would be used to create the practice problems "7 TIMES 3", "7 TIMES 4", "7 TIMES 5", and so on.

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

The following examples illustrate the behavior of the `MultiPractice` 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
<code>StudyPractice p1 = new MultiPractice(7, 3);</code> <code>System.out.println(p1.getProblem());</code>	7 TIMES 3
<code>p1.nextProblem();</code> <code>System.out.println(p1.getProblem());</code>	7 TIMES 4
<code>p1.nextProblem();</code> <code>System.out.println(p1.getProblem());</code>	7 TIMES 5
<code>p1.nextProblem();</code> <code>System.out.println(p1.getProblem());</code>	7 TIMES 6

Example 2

Code segment	Output produced
<code>StudyPractice p2 = new MultiPractice(4, 12);</code> <code>p2.nextProblem();</code> <code>System.out.println(p2.getProblem());</code> <code>System.out.println(p2.getProblem());</code>	4 TIMES 13 4 TIMES 13
<code>p2.nextProblem();</code> <code>p2.nextProblem();</code> <code>System.out.println(p2.getProblem());</code>	4 TIMES 15
<code>p2.nextProblem();</code> <code>System.out.println(p2.getProblem());</code>	4 TIMES 16

Question 2 continues on page 10.

Interface information for this question

```
public interface StudyPractice
```

```
String getProblem()
void nextProblem()
```

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

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

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

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

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

```
HiddenWord puzzle = new HiddenWord("HARPS");
```

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

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

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

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

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

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

Write the complete `NumberGroup` interface. It must have exactly one method.

Part (b) begins on page 17.

- (b) A *range* represents a number group that contains all (and only) the integers between a minimum value and a maximum value, inclusive.

Write the `Range` class, which is a `NumberGroup`. The `Range` class represents the group of `int` values that range from a given minimum value up through a given maximum value, inclusive. For example, the declaration

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

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

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

Part (c) begins on page 18.

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

```
private List<NumberGroup> groupList;
```

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

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

Call	Result
<code>multiple1.contains(2)</code>	<code>true</code>
<code>multiple1.contains(9)</code>	<code>false</code>
<code>multiple1.contains(6)</code>	<code>true</code>

Complete method `contains` below.

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

STOP

END OF EXAM

2. This question involves reasoning about the GridWorld case study. Reference materials are provided in the appendixes.

A `Director` is a type of `Rock` that has the following characteristics.

- A `Director` has an initial color of `Color.RED` and alternates between `Color.RED` and `Color.GREEN` each time it acts.
- If the color of a `Director` is `Color.GREEN` when it begins to act, it will cause any `Actor` objects in its neighboring cells to turn 90 degrees to their right.

Write the complete `Director` class, including the zero-parameter constructor and any necessary instance variables and methods. Assume that the `Color` class has been imported.

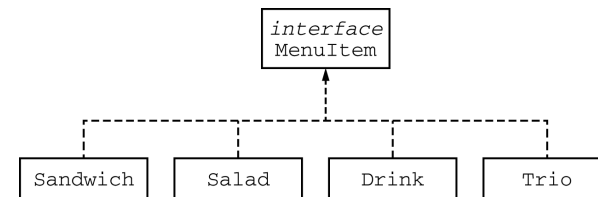
4. The menu at a lunch counter includes a variety of sandwiches, salads, and drinks. The menu also allows a customer to create a "trio," which consists of three menu items: a sandwich, a salad, and a drink. The price of the trio is the sum of the two highest-priced menu items in the trio; one item with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

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

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

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

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



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

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

Question 4 continues on page 13

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

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

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

```
Sandwich sandwich;  
Salad salad;  
Drink drink;  
/* Code that initializes sandwich, salad, and drink */  
  
Trio trio = new Trio(sandwich, salad, drink); // Compiles without error  
  
Trio trio1 = new Trio(salad, sandwich, drink); // Compile-time error  
Trio trio2 = new Trio(sandwich, salad, salad); // Compile-time error
```

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Write the complete `Trio` class below.

STOP
END OF EXAM