

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>	"0-0-Red"	
<code>game.recordPlay(1);</code>		Team 1 earns 1 point because the game always begins with team 1 as the active team.
<code>info = game.getScore();</code>	"1-0-Red"	
<code>game.recordPlay(0);</code>		Team 1's play failed, so team 2 is now active.
<code>info = game.getScore();</code>	"1-0-Blue"	
<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>		Team 2 earns 3 points.
<code>info = game.getScore();</code>	"1-3-Blue"	
<code>game.recordPlay(1);</code>		Team 2 earns 1 point.
<code>game.recordPlay(0);</code>		Team 2's play failed, so team 1 is now active.
<code>info = game.getScore();</code>	"1-4-Red"	
<code>game.recordPlay(0);</code>		Team 1's play failed, so team 2 is now active.
<code>game.recordPlay(4);</code>		Team 2 earns 4 points.
<code>game.recordPlay(0);</code>		Team 2's play failed, so team 1 is now active.
<code>info = game.getScore();</code>	"1-8-Red"	
<code>Scoreboard match = new Scoreboard("Lions", "Tigers");</code>		match is a new and independent <code>Scoreboard</code> object.
<code>info = match.getScore();</code>	"0-0-Lions"	
<code>info = game.getScore();</code>	"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.

1. This question involves the `AppointmentBook` class, which provides methods for students to schedule appointments with their teacher. Appointments can be scheduled during one of eight class periods during the school day, numbered 1 through 8. A requested appointment has a duration, which is the number of minutes the appointment will last. The 60 minutes within a period are numbered 0 through 59. In order for an appointment to be scheduled, the teacher must have a block of consecutive, available minutes that contains at least the requested number of minutes in a requested period. Scheduled appointments must start and end within the same period.

The AppointmentBook class contains two helper methods, isMinuteFree and reserveBlock. You will write two additional methods of the AppointmentBook class.

```
public class AppointmentBook
{
    /**
     * Returns true if minute in period is available for an appointment and returns
     * false otherwise
     * Preconditions: 1 <= period <= 8; 0 <= minute <= 59
     */
    private boolean isMinuteFree(int period, int minute)
    { /* implementation not shown */ }

    /**
     * Marks the block of minutes that starts at startMinute in period and
     * is duration minutes long as reserved for an appointment
     * Preconditions: 1 <= period <= 8; 0 <= startMinute <= 59;
     * 1 <= duration <= 60
     */
    private void reserveBlock(int period, int startMinute, int duration)
    { /* implementation not shown */ }

    /**
     * Searches for the first block of duration free minutes during period, as described in
     * part (a). Returns the first minute in the block if such a block is found or returns -1 if no
     * such block is found.
     * Preconditions: 1 <= period <= 8; 1 <= duration <= 60
     */
    public int findFreeBlock(int period, int duration)
    { /* to be implemented in part (a) */ }

    /**
     * Searches periods from startPeriod to endPeriod, inclusive, for a block
     * of duration free minutes, as described in part (b). If such a block is found,
     * calls reserveBlock to reserve the block of minutes and returns true; otherwise
     * returns false.
     * Preconditions: 1 <= startPeriod <= endPeriod <= 8; 1 <= duration <= 60
     */
    public boolean makeAppointment(int startPeriod, int endPeriod,
                                   int duration)
    { /* to be implemented in part (b) */ }

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

- (a) Write the `findFreeBlock` method, which searches `period` for the first block of free minutes that is `duration` minutes long. If such a block is found, `findFreeBlock` returns the first minute in the block. Otherwise, `findFreeBlock` returns `-1`. The `findFreeBlock` method uses the helper method `isMinuteFree`, which returns `true` if a particular minute is available to be included in a new appointment and returns `false` if the minute is unavailable.

Consider the following list of unavailable and available minutes in period 2.

Minutes in Period 2	Available?
0–9 (10 minutes)	No
10–14 (5 minutes)	Yes
15–29 (15 minutes)	No
30–44 (15 minutes)	Yes
45–49 (5 minutes)	No
50–59 (10 minutes)	Yes

The method call `findFreeBlock(2, 15)` would return 30 to indicate that a 15-minute block starting with minute 30 is available. No steps should be taken as a result of the call to `findFreeBlock` to mark those 15 minutes as unavailable.

The method call `findFreeBlock(2, 9)` would also return 30. Whenever there are multiple blocks that satisfy the requirement, the earliest starting minute is returned.

The method call `findFreeBlock(2, 20)` would return `-1`, since no 20-minute block of available minutes exists in period 2.

Complete method `findFreeBlock`. You must use `isMinuteFree` appropriately in order to receive full credit.

```
/**
 * Searches for the first block of duration free minutes during period, as described in
 * part (a). Returns the first minute in the block if such a block is found or returns -1 if no
 * such block is found.
 * Preconditions: 1 <= period <= 8; 1 <= duration <= 60
 */
public int findFreeBlock(int period, int duration)
```

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.

- (b) Write the `makeAppointment` method, which searches the periods from `startPeriod` to `endPeriod`, inclusive, for the earliest block of `duration` available minutes in the lowest-numbered period. If such a block is found, the `makeAppointment` method calls the helper method `reserveBlock` to mark the minutes in the block as unavailable and returns `true`. If no such block is found, the `makeAppointment` method returns `false`.

Consider the following list of unavailable and available minutes in periods 2, 3, and 4 and three successive calls to `makeAppointment`.

Period	Minutes	Available?
2	0–24 (25 minutes)	No
2	25–29 (5 minutes)	Yes
2	30–59 (30 minutes)	No
3	0–14 (15 minutes)	Yes
3	15–40 (26 minutes)	No
3	41–59 (19 minutes)	Yes
4	0–4 (5 minutes)	No
4	5–29 (25 minutes)	Yes
4	30–43 (14 minutes)	No
4	44–59 (16 minutes)	Yes

The method call `makeAppointment(2, 4, 22)` returns `true` and results in the minutes 5 through 26, inclusive, in period 4 being marked as unavailable.

The method call `makeAppointment(3, 4, 3)` returns `true` and results in the minutes 0 through 2, inclusive, in period 3 being marked as unavailable.

The method call `makeAppointment(2, 4, 30)` returns `false`, since there is no block of 30 available minutes in periods 2, 3, or 4.

The following shows the updated list of unavailable and available minutes in periods 2, 3, and 4 after the three example method calls are complete.

Period	Minutes	Available?
2	0–24 (25 minutes)	No
2	25–29 (5 minutes)	Yes
2	30–59 (30 minutes)	No
3	0–2 (3 minutes)	No
3	3–14 (12 minutes)	Yes
3	15–40 (26 minutes)	No
3	41–59 (19 minutes)	Yes
4	0–26 (27 minutes)	No
4	27–29 (3 minutes)	Yes
4	30–43 (14 minutes)	No
4	44–59 (16 minutes)	Yes

Complete method `makeAppointment`. Assume that `findFreeBlock` works as intended, regardless of what you wrote in part (a). You must use `findFreeBlock` and `reserveBlock` appropriately in order to receive full credit.

```
/**
 * Searches periods from startPeriod to endPeriod, inclusive, for a block
 * of duration free minutes, as described in part (b). If such a block is found,
 * calls reserveBlock to reserve the block of minutes and returns true; otherwise
 * returns false.
 * Preconditions: 1 <= startPeriod <= endPeriod <= 8; 1 <= duration <= 60
 */
public boolean makeAppointment(int startPeriod, int endPeriod,
                               int duration)
```

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.

Class information for this question

```
public class AppointmentBook
private boolean isMinuteFree(int period, int minute)
private void reserveBlock(int period, int startMinute,
                           int duration)
public int findFreeBlock(int period, int duration)
public boolean makeAppointment(int startPeriod, int endPeriod,
                               int duration)
```

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
<code>Textbook bio2015 = new Textbook("Biology", 49.75, 2);</code>		bio2015 is a <code>Textbook</code> with a title of "Biology", a price of 49.75, and an edition of 2.
<code>Textbook bio2019 = new Textbook("Biology", 39.75, 3);</code>		bio2019 is a <code>Textbook</code> with a title of "Biology", a price of 39.75, and an edition of 3.
<code>bio2019.getEdition();</code>	3	The edition is returned.
<code>bio2019.getBookInfo();</code>	"Biology-39.75-3"	The formatted string containing the title, price, and edition of bio2019 is returned.
<code>bio2019. canSubstituteFor(bio2015);</code>	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.
<code>bio2015. canSubstituteFor(bio2019);</code>	false	bio2015 is not a valid substitute for bio2019, since the edition of bio2015 is less than the edition of bio2019.
<code>Textbook math = new Textbook("Calculus", 45.25, 1);</code>		math is a <code>Textbook</code> with a title of "Calculus", a price of 45.25, and an edition of 1.
<code>bio2015. canSubstituteFor(math);</code>	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.

1. The `APCalendar` class contains methods used to calculate information about a calendar. You will write two methods of the class.

```
public class APCalendar
{
    /** Returns true if year is a leap year and false otherwise. */
    private static boolean isLeapYear(int year)
    { /* implementation not shown */ }

    /** Returns the number of leap years between year1 and year2, inclusive.
     * Precondition: 0 <= year1 <= year2
     */
    public static int numberOfLeapYears(int year1, int year2)
    { /* to be implemented in part (a) */ }

    /** Returns the value representing the day of the week for the first day of year,
     * where 0 denotes Sunday, 1 denotes Monday, ..., and 6 denotes Saturday.
     */
    private static int firstDayOfYear(int year)
    { /* implementation not shown */ }

    /** Returns n, where month, day, and year specify the nth day of the year.
     * Returns 1 for January 1 (month = 1, day = 1) of any year.
     * Precondition: The date represented by month, day, year is a valid date.
     */
    private static int dayOfYear(int month, int day, int year)
    { /* implementation not shown */ }

    /** Returns the value representing the day of the week for the given date
     * (month, day, year), where 0 denotes Sunday, 1 denotes Monday, ...,
     * and 6 denotes Saturday.
     * Precondition: The date represented by month, day, year is a valid date.
     */
    public static int dayOfWeek(int month, int day, int year)
    { /* to be implemented in part (b) */ }

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

- (a) Write the static method `numberOfLeapYears`, which returns the number of leap years between `year1` and `year2`, inclusive.

In order to calculate this value, a helper method is provided for you.

- `isLeapYear(year)` returns `true` if `year` is a leap year and `false` otherwise.

Complete method `numberOfLeapYears` below. You must use `isLeapYear` appropriately to receive full credit.

```
/** Returns the number of leap years between year1 and year2, inclusive.
 * Precondition: 0 <= year1 <= year2
 */
public static int numberOfLeapYears(int year1, int year2)
```

- (b) Write the static method `dayOfWeek`, which returns the integer value representing the day of the week for the given date (`month`, `day`, `year`), where 0 denotes Sunday, 1 denotes Monday, ..., and 6 denotes Saturday. For example, 2019 began on a Tuesday, and January 5 is the fifth day of 2019. As a result, January 5, 2019, fell on a Saturday, and the method call `dayOfWeek(1, 5, 2019)` returns 6.

As another example, January 10 is the tenth day of 2019. As a result, January 10, 2019, fell on a Thursday, and the method call `dayOfWeek(1, 10, 2019)` returns 4.

In order to calculate this value, two helper methods are provided for you.

- `firstDayOfYear(year)` returns the integer value representing the day of the week for the first day of year, where 0 denotes Sunday, 1 denotes Monday, ..., and 6 denotes Saturday. For example, since 2019 began on a Tuesday, `firstDayOfYear(2019)` returns 2.
- `dayOfYear(month, day, year)` returns n , where `month`, `day`, and `year` specify the n th day of the year. For the first day of the year, January 1 (`month` = 1, `day` = 1), the value 1 is returned. This method accounts for whether `year` is a leap year. For example, `dayOfYear(3, 1, 2017)` returns 60, since 2017 is not a leap year, while `dayOfYear(3, 1, 2016)` returns 61, since 2016 is a leap year.

Class information for this question

```
public class APCalendar
```

```
private static boolean isLeapYear(int year)
public static int numberOfLeapYears(int year1, int year2)
private static int firstDayOfYear(int year)
private static int dayOfYear(int month, int day, int year)
public static int dayOfWeek(int month, int day, int year)
```

Complete method `dayOfWeek` below. You must use `firstDayOfYear` and `dayOfYear` appropriately to receive full credit.

```
/** Returns the value representing the day of the week for the given date
 * (month, day, year), where 0 denotes Sunday, 1 denotes Monday, ...,
 * and 6 denotes Saturday.
 * Precondition: The date represented by month, day, year is a valid date.
 */
public static int dayOfWeek(int month, int day, int year)
```

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 `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
<code>StudyPractice p1 = new MultPractice(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 MultPractice(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.

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"`

<code>"disk"</code>
<code>"error on disk"</code>
<code>"error on /dev/disk disk"</code>
<code>"error on disk DSK1"</code>

Descriptions that do not properly contain `"disk"`

<code>"DISK"</code>
<code>"error on disk3"</code>
<code>"error on /dev/disk"</code>
<code>"diskette"</code>

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)
```

4. This question involves the process of taking a list of words, called `wordList`, and producing a formatted string of a specified length. The list `wordList` contains at least two words, consisting of letters only.

When the formatted string is constructed, spaces are placed in the gaps between words so that as many spaces as possible are evenly distributed to each gap. The equal number of spaces inserted into each gap is referred to as the *basic gap width*. Any *leftover spaces* are inserted one at a time into the gaps from left to right until there are no more leftover spaces.

The following three examples illustrate these concepts. In each example, the list of words is to be placed into a formatted string of length 20.

Example 1: `wordList`: ["AP", "COMP", "SCI", "ROCKS"]

Total number of letters in words: 14

Number of gaps between words: 3

Basic gap width: 2

Leftover spaces: 0

Formatted string:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19																					
	A		P						C		O		M		P						S		C		I						R		O		C		K		S	

Example 2: `wordList`: ["GREEN", "EGGS", "AND", "HAM"]

Total number of letters in words: 15

Number of gaps between words: 3

Basic gap width: 1

Leftover spaces: 2

The leftover spaces are inserted one at a time between the words from left to right until there are no more leftover spaces. In this example, the first two gaps get an extra space.

Formatted string:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19																					
	G		R		E		E		N						E		G		G		S						A		N		D				H		A		M	

Example 3: `wordList`: ["BEACH", "BALL"]

Total number of letters in words: 9

Number of gaps between words: 1

Basic gap width: 11

Leftover spaces: 0

Formatted string:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19																			
	B		E		A		C		H																						B		A		L		L	

You will implement three `static` methods in a class named `StringFormatter` that is not shown.

- (a) Write the `StringFormatter` method `totalLetters`, which returns the total number of letters in the words in its parameter `wordList`. For example, if the variable `List<String> words` is `["A", "frog", "is"]`, then the call `StringFormatter.totalLetters(words)` returns 7. You may assume that all words in `wordList` consist of one or more letters.

Complete method `totalLetters` below.

```
/** Returns the total number of letters in wordList.  
 * Precondition: wordList contains at least two words, consisting of letters only.  
 */  
public static int totalLetters(List<String> wordList)
```

Part (b) begins on page 20.

- (b) Write the `StringFormatter` method `basicGapWidth`, which returns the basic gap width as defined earlier.

Class information for this question

```
public class StringFormatter

public static int totalLetters(List<String> wordList)
public static int basicGapWidth(List<String> wordList,
                                int formattedLen)
public static int leftoverSpaces(List<String> wordList,
                                int formattedLen)
public static String format(List<String> wordList, int formattedLen)
```

WRITE YOUR SOLUTION ON THE NEXT PAGE.

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

Complete method `basicGapWidth` below.

```
/** Returns the basic gap width when wordList is used to produce
 * a formatted string of formattedLen characters.
 * Precondition: wordList contains at least two words, consisting of letters only.
 * formattedLen is large enough for all the words and gaps.
 */
public static int basicGapWidth(List<String> wordList,
                                int formattedLen)
```

Part (c) begins on page 22.

- (c) Write the `StringFormatter` method `format`, which returns the formatted string as defined earlier. The `StringFormatter` class also contains a method called `leftoverSpaces`, which has already been implemented. This method returns the number of leftover spaces as defined earlier and is shown below.

```
/** Returns the number of leftover spaces when wordList is used to produce
 * a formatted string of formattedLen characters.
 * Precondition: wordList contains at least two words, consisting of letters only.
 * formattedLen is large enough for all the words and gaps.
 */
public static int leftoverSpaces(List<String> wordList,
                                int formattedLen)
{ /* implementation not shown */ }
```

Class information for this question

```
public class StringFormatter

public static int totalLetters(List<String> wordList)
public static int basicGapWidth(List<String> wordList,
                                int formattedLen)
public static int leftoverSpaces(List<String> wordList,
                                int formattedLen)
public static String format(List<String> wordList, int formattedLen)
```

WRITE YOUR SOLUTION ON THE NEXT PAGE.

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

Complete method `format` below.

```
/** Returns a formatted string consisting of the words in wordList separated by spaces.
 * Precondition: The wordList contains at least two words, consisting of letters only.
 * formattedLen is large enough for all the words and gaps.
 * Postcondition: All words in wordList appear in the formatted string.
 * - The words appear in the same order as in wordList.
 * - The number of spaces between words is determined by basicGapWidth and the
 * distribution of leftoverSpaces from left to right, as described in the question.
 */
public static String format(List<String> wordList, int formattedLen)
```

STOP

END OF EXAM

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