1. This question involves simulation of the play and scoring of a single-player video game. In the game, a player attempts to complete three levels. A level in the game is represented by the `Level` class.

```
public class Level
{
   /** Returns true if the player reached the goal on this level and returns false otherwise */
   public boolean goalReached()
   {  /* implementation not shown */  }

   /** Returns the number of points (a positive integer) recorded for this level */
   public int getPoints()
   {  /* implementation not shown */  }

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

Play of the game is represented by the `Game` class. You will write two methods of the `Game` class.

```
public class Game
{
   private Level levelOne;
   private Level levelTwo;
   private Level levelThree;

   /** Postcondition: All instance variables have been initialized. */
   public Game()
   {  /* implementation not shown */  }

   /** Returns true if this game is a bonus game and returns false otherwise */
   public boolean isBonus()
   {  /* implementation not shown */  }

   /** Simulates the play of this Game (consisting of three levels) and updates all relevant
    *  game data
    */
   public void play()
   {  /* implementation not shown */  }

   /** Returns the score earned in the most recently played game, as described in part (a) */
   public int getScore()
   {  /* to be implemented in part (a) */  }

   /** Simulates the play of num games and returns the highest score earned, as
    *  described in part (b)
    *  Precondition: num > 0
    */
   public int playManyTimes(int num)
   {  /* to be implemented in part (b) */  }

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

(a) Write the `getScore` method, which returns the score for the most recently played game. Each game consists of three levels. The score for the game is computed using the following helper methods.

- The `isBonus` method of the `Game` class returns `true` if this is a bonus game and returns `false` otherwise.
- The `goalReached` method of the `Level` class returns `true` if the goal has been reached on a particular level and returns `false` otherwise.
- The `getPoints` method of the `Level` class returns the number of points recorded on a particular level. Whether or not recorded points are earned (included in the game score) depends on the rules of the game, which follow.

The score for the game is computed according to the following rules.

- Level one points are earned only if the level one goal is reached. Level two points are earned only if both the level one and level two goals are reached. Level three points are earned only if the goals of all three levels are reached.
- The score for the game is the sum of the points earned for levels one, two, and three.
- If the game is a bonus game, the score for the game is tripled.

```
public int getScore() {
    int score = 0;
    if (levelOne.goalReached() == true) {
        score = levelOne.getPoints();
        if (levelTwo.goalReached() == true) {
            score += levelTwo.getPoints();
        } if (levelThree.goalReached() == true) {
            score += levelThree.getPoints();
        }
    }
    if (isBonus() == true) {
        score *= 3;
    }
    return score;
}
```

GO ON TO THE NEXT PAGE.

The following table shows some examples of game score calculations.

	Level One Results	Level Two Results	Level Three Results	`isBonus` Return Value	Score Calculation
`goalReached` **Return Value:** `getPoints` **Return Value:**	`true` `200`	`true` `100`	`true` `500`	`true`	(200 + 100 + 500) × 3 = 2,400 The recorded points for levels one, two, and three are earned because the goals were reached in all three levels. The earned points are multiplied by 3 because `isBonus` returns `true`.
`goalReached` **Return Value:** `getPoints` **Return Value:**	`true` `200`	`true` `100`	`false` `500`	`false`	200 + 100 = 300 The recorded points for level one and level two are earned because the goal was reached in levels one and two. The recorded points for level three are not earned because the goal was not reached in level three.
`goalReached` **Return Value:** `getPoints` **Return Value:**	`true` `200`	`false` `100`	`true` `500`	`true`	200 × 3 = 600 The recorded points for only level one are earned because the goal was not reached in level two. The earned points are multiplied by 3 because `isBonus` returns `true`.
`goalReached` **Return Value:** `getPoints` **Return Value:**	`false` `200`	`true` `100`	`true` `500`	`false`	0 Because the goal in level one was not reached, no points are earned for any level.

Complete the `getScore` method.

```
/** Returns the score earned in the most recently played game, as described in part (a) */
public int getScore()
```

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.

(b) Write the `playManyTimes` method, which simulates the play of `num` games and returns the highest game score earned. For example, if the four plays of the game that are simulated as a result of the method call `playManyTimes(4)` earn scores of `75`, `50`, `90`, and `20`, then the method should return `90`.

Play of the game is simulated by calling the helper method `play`. Note that if `play` is called only one time followed by multiple consecutive calls to `getScore`, each call to `getScore` will return the score earned in the single simulated play of the game.

Complete the `playManyTimes` method. Assume that `getScore` works as intended, regardless of what you wrote in part (a). You must call `play` and `getScore` appropriately in order to receive full credit.

```
/** Simulates the play of num games and returns the highest score earned, as
 *  described in part (b)
 *  Precondition: num > 0
 */
public int playManyTimes(int num)
```

```
public int playManyTimes(int num) {
    int max = 0;
    for (int i = 0; i < num; i++) {
        play();
        int score = getScore();
        if (score > max) {
            max = score;
        }
    }
    return max;
}
```

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`		`math` is a `Textbook` with a

```java
public class Textbook extends Book {
    private int edition;

    public Textbook(String tbTitle, double tbPrice, int tbEdition) {
        super(tbTitle, tbPrice);
        edition = tbEdition;
    }

    public int getEdition() {
        return edition;
    }

    public boolean canSubstituteFor (Textbook other) {
        return other.getTitle().equals(getTitle()) && edition >= other.getEdition();
    }

    public String getBookInfo() {
        return super.getBookInfo() + "-" + edition;
    }

}
```

3. Users of a website are asked to provide a review of the website at the end of each visit. Each review, represented by an object of the `Review` class, consists of an integer indicating the user's rating of the website and an optional `String` comment field. The comment field in a `Review` object ends with a period (`"."`), exclamation point (`"!"`), or letter, or is a `String` of length `0` if the user did not enter a comment.

```
public class Review
{
   private int rating;
   private String comment;

   /** Precondition: r >= 0
    *      c is not null.
    */
   public Review(int r, String c)
   {
      rating = r;
      comment = c;
   }

   public int getRating()
   {
      return rating;
   }

   public String getComment()
   {
      return comment;
   }

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

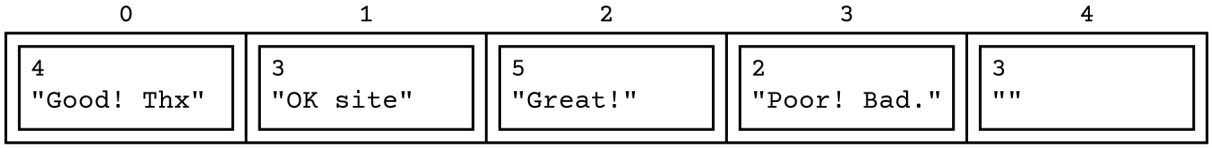
GO ON TO THE NEXT PAGE.

The `ReviewAnalysis` class contains methods used to analyze the reviews provided by users. You will write two methods of the `ReviewAnalysis` class.

```
public class ReviewAnalysis
{
    /** All user reviews to be included in this analysis */
    private Review[] allReviews;

    /** Initializes allReviews to contain all the Review objects to be analyzed */
    public ReviewAnalysis()
    {  /* implementation not shown */  }

    /** Returns a double representing the average rating of all the Review objects to be
     *  analyzed, as described in part (a)
     *  Precondition: allReviews contains at least one Review.
     *      No element of allReviews is null.
     */
    public double getAverageRating()
    {  /* to be implemented in part (a) */  }

    /** Returns an ArrayList of String objects containing formatted versions of
     *  selected user comments, as described in part (b)
     *  Precondition: allReviews contains at least one Review.
     *      No element of allReviews is null.
     *  Postcondition: allReviews is unchanged.
     */
    public ArrayList<String> collectComments()
    {  /* to be implemented in part (b) */  }
}
```

```
public double getAverageRating() {
    int sum = 0;
    for (Review r : allReviews) {
        sum += r.getRating();
    }
    return (double) sum / allReviews.length();
}
```

(a) Write the `ReviewAnalysis` method `getAverageRating`, which returns the average rating (arithmetic mean) of all elements of `allReviews`. For example, `getAverageRating` would return `3.4` if `allReviews` contained the following `Review` objects.



Complete method `getAverageRating`.

```
/** Returns a double representing the average rating of all the Review objects to be
 *  analyzed, as described in part (a)
 *  Precondition: allReviews contains at least one Review.
 *       No element of allReviews is null.
 */
public double getAverageRating()
```

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.

Class information for this question

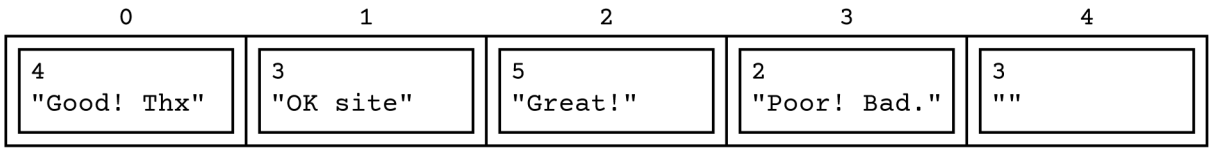
```
public class Review

private int rating
private String comment

public Review(int r, String c)
public int getRating()
public String getComment()

public class ReviewAnalysis

private Review[] allReviews

public ReviewAnalysis()
public double getAverageRating()
public ArrayList<String> collectComments()
```

GO ON TO THE NEXT PAGE.

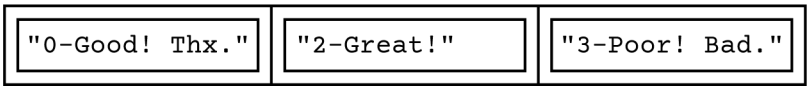
(b) Write the `ReviewAnalysis` method `collectComments`, which collects and formats only comments that contain an exclamation point. The method returns an `ArrayList` of `String` objects containing copies of user comments from `allReviews` that contain an exclamation point, formatted as follows. An empty `ArrayList` is returned if no comment in `allReviews` contains an exclamation point.

- The `String` inserted into the `ArrayList` to be returned begins with the index of the `Review` in `allReviews`.
- The index is immediately followed by a hyphen (`"-"`).
- The hyphen is followed by a copy of the original comment.
- The `String` must end with either a period or an exclamation point. If the original comment from `allReviews` does not end in either a period or an exclamation point, a period is added.

The following example of `allReviews` is repeated from part (a).



The following `ArrayList` would be returned by a call to `collectComments` with the given contents of `allReviews`. The reviews at index `1` and index `4` in `allReviews` are not included in the `ArrayList` to return since neither review contains an exclamation point.



Complete method `collectComments`.

```
/** Returns an ArrayList of String objects containing formatted versions of
 *  selected user comments, as described in part (b)
 *  Precondition: allReviews contains at least one Review.
 *                No element of allReviews is null.
```

```
public ArrayList<String> collectComments() {
    ArrayList<String> commentList = new ArrayList<String>();
    for (int i = 0; i < allReviews.length; i++) {
        String comment = allReviews[i].getComment();
        if (comment.indexOf(str:"!") >= 0) {
            String last = comment.substring(comment.length() - 1);
            if (!last.equals(anObject:"!") && !last.equals(anObject:".")) {
                comment += ".";
            }
            commentList.add(i + "-" + comment);
        }
    }
    return commentList;
}
```

4. This question involves a two-dimensional array of integers that represents a collection of randomly generated data. A partial declaration of the `Data` class is shown. You will write two methods of the `Data` class.

```
public class Data
{
   public static final int MAX = /* value not shown */;
   private int[][] grid;

   /** Fills all elements of grid with randomly generated values, as described in part (a)
    *  Precondition: grid is not null.
    *       grid has at least one element.
    */
   public void repopulate()
   {  /* to be implemented in part (a) */  }

   /** Returns the number of columns in grid that are in increasing order, as described
    *  in part (b)
    *  Precondition: grid is not null.
    *       grid has at least one element.
    */
   public int countIncreasingCols()
   {  /* to be implemented in part (b) */  }

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

GO ON TO THE NEXT PAGE.

(a) Write the `repopulate` method, which assigns a newly generated random value to each element of `grid`. Each value is computed to meet all of the following criteria, and all valid values must have an equal chance of being generated.

- The value is between 1 and `MAX`, inclusive.
- The value is divisible by 10.
- The value is not divisible by 100.

Complete the `repopulate` method.

```
/** Fills all elements of grid with randomly generated values, as described in part (a)
 *  Precondition: grid is not null.
 *      grid has at least one element.
 */
public void repopulate()
```

```java
public void repopulate() {
    for (int row = 0; row < grid.length; row++) {
        for (int col = 0; col < grid[0].length; col++) {
            int rval = (int) (Math.random() * MAX) + 1;
            while (rval % 10 != 0 || rval % 100 == 0) {
                rval = (int) (Math.random() * MAX) + 1;
            }
            grid[row][col] = rval;
        }
    }
}
```

(b) Write the `countIncreasingCols` method, which returns the number of columns in `grid` that are in increasing order. A column is considered to be in increasing order if the element in each row after the first row is greater than or equal to the element in the previous row. A column with only one row is considered to be in increasing order.

The following examples show the `countIncreasingCols` return values for possible contents of `grid`.

The return value for the following contents of `grid` is `1`, since the first column is in increasing order but the second and third columns are not.

10	50	40
20	40	20
30	50	30

The return value for the following contents of `grid` is `2`, since the first and third columns are in increasing order but the second and fourth columns are not.

10	540	440	440
220	450	440	190

Complete the `countIncreasingCols` method.

```
/** Returns the number of columns in grid that are in increasing order, as described
 *  in part (b)
 *  Precondition: grid is not null.
 *      grid has at least one element.
 */
public int countIncreasingCols()
```

```java
public int countIncreasingCols() {
    int count = 0;
    for (int col = 0; col < grid[0].length; col++) {
        boolean isOrdered = true;
        for (int row = 1; row < grid.length; row++) {
            if (grid[row][col] <= grid[row - 1][col]) {
                isOrdered = false;
            }
        }
        if (isOrdered) {
            count++;
        }
    }
    return count;
}
```