Applying the Scoring Criteria

Apply the question scoring criteria first, which always takes precedence. Penalty points can only be deducted in a part of the question that has earned credit via the question rubric. No part of a question (a, b, c) may have a negative point total. A given penalty can be assessed only once for a question, even if it occurs multiple times or in multiple parts of that question. A maximum of 3 penalty points may be assessed per question.

1-Point Penalty

v) Array/collection access confusion (`[] get`)

w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check)

x) Local variables used but none declared

y) Destruction of persistent data (e.g., changing value referenced by parameter)

z) Void method or constructor that returns a value

No Penalty

- Extraneous code with no side-effect (e.g., valid precondition check, no-op)
- Spelling/case discrepancies where there is no ambiguity*
- Local variable not declared provided other variables are declared in some part
- `private` or `public` qualifier on a local variable
- Missing `public` qualifier on class or constructor header
- Keyword used as an identifier
- Common mathematical symbols used for operators (× • ÷ ≤ ≥ <> ≠)
- `[]` vs. `()` vs. `<>`
- `=` instead of `==` and vice versa
- `length/size` confusion for array, `String`, `List`, or `ArrayList`; with or without `( )`
- Extraneous `[]` when referencing entire array
- `[i,j]` instead of `[i][j]`
- Extraneous size in array declaration, e.g., `int[size] nums = new int[size];`
- Missing `;` where structure clearly conveys intent
- Missing `{ }` where indentation clearly conveys intent
- Missing `( )` on parameter-less method or constructor invocations
- Missing `( )` around `if` or `while` conditions

Spelling and case discrepancies for identifiers fall under the "No Penalty" category only if the correction can be **unambiguously inferred from context, for example, "ArayList" instead of "ArrayList". As a counterexample, note that if the code declares "`int G=99, g=0;`", then uses "`while (G < 10)`" instead of "`while (g < 10)`", the context does **not** allow for the reader to assume the use of the lower case variable.*

Basic Part FRQ

1. 2019(1). 1

1. A mathematical sequence is an ordered list of numbers. This question involves a sequence called a *hailstone sequence*. If n is the value of a term in the sequence, then the following rules are used to find the next term, if one exists.

   - If n is 1, the sequence terminates.
   - If n is even, then the next term is $\frac{n}{2}$.
   - If n is odd, then the next term is $3n + 1$.

   For this question, assume that when the rules are applied, the sequence will eventually terminate with the term $n = 1$.

   The following are examples of hailstone sequences.

   Example 1: 5, 16, 8, 4, 2, 1

   - The first term is 5, so the second term is $5 * 3 + 1 = 16$.
   - The second term is 16, so the third term is $\frac{16}{2} = 8$.
   - The third term is 8, so the fourth term is $\frac{8}{2} = 4$.
   - The fourth term is 4, so the fifth term is $\frac{4}{2} = 2$.
   - The fifth term is 2, so the sixth term is $\frac{2}{2} = 1$.
   - Since the sixth term is 1, the sequence terminates.

   Example 2: 8, 4, 2, 1

   - The first term is 8, so the second term is $\frac{8}{2} = 4$.
   - The second term is 4, so the third term is $\frac{4}{2} = 2$.
   - The third term is 2, so the fourth term is $\frac{2}{2} = 1$.
   - Since the fourth term is 1, the sequence terminates.

The Hailstone class, shown below, is used to represent a hailstone sequence. You will write three methods in the Hailstone class.

```
public class Hailstone
{
   /** Returns the length of a hailstone sequence that starts with n,
    *  as described in part (a).
    *  Precondition: n > 0
    */
   public static int hailstoneLength(int n)
   {  /* to be implemented in part (a) */  }

   /** Returns true if the hailstone sequence that starts with n is considered long
    *  and false otherwise, as described in part (b).
    *  Precondition: n > 0
    */
   public static boolean isLongSeq(int n)
   {  /* to be implemented in part (b) */  }

   /** Returns the proportion of the first n hailstone sequences that are considered long,
    *  as described in part (c).
    *  Precondition: n > 0
    */
   public static double propLong(int n)
   {  /* to be implemented in part (c) */  }

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

(a) The length of a hailstone sequence is the number of terms it contains. For example, the hailstone sequence in example 1 (5, 16, 8, 4, 2, 1) has a length of 6 and the hailstone sequence in example 2 (8, 4, 2, 1) has a length of 4.

Write the method `hailstoneLength(int n)`, which returns the length of the hailstone sequence that starts with `n`.

```
/** Returns the length of a hailstone sequence that starts with n, as described in part (a).
 *  Precondition: n > 0
 */
public static int hailstoneLength(int n)
```

Class information for this question

```
public class Hailstone

public static int hailstoneLength(int n)
public static boolean isLongSeq(int n)
public static double propLong(int n)
```

(b) A hailstone sequence is considered long if its length is greater than its starting value. For example, the hailstone sequence in example 1 (5, 16, 8, 4, 2, 1) is considered long because its length (6) is greater than its starting value (5). The hailstone sequence in example 2 (8, 4, 2, 1) is not considered long because its length (4) is less than or equal to its starting value (8).

Write the method `isLongSeq(int n)`, which returns `true` if the hailstone sequence starting with `n` is considered long and returns `false` otherwise. Assume that `hailstoneLength` works as intended, regardless of what you wrote in part (a). You must use `hailstoneLength` appropriately to receive full credit.

```
/** Returns true if the hailstone sequence that starts with n is considered long
 *  and false otherwise, as described in part (b).
 *  Precondition: n > 0
 */
public static boolean isLongSeq(int n)
```

(c) The method `propLong(int n)` returns the proportion of long hailstone sequences with starting values between 1 and `n`, inclusive.

Consider the following table, which provides data about the hailstone sequences with starting values between 1 and 10, inclusive.

Starting Value	**Terms in the Sequence**	**Length of the Sequence**	**Long?**
1	1	1	No
2	2, 1	2	No
3	3, 10, 5, 16, 8, 4, 2, 1	8	Yes
4	4, 2, 1	3	No
5	5, 16, 8, 4, 2, 1	6	Yes
6	6, 3, 10, 5, 16, 8, 4, 2, 1	9	Yes
7	7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1	17	Yes
8	8, 4, 2, 1	4	No
9	9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1	20	Yes
10	10, 5, 16, 8, 4, 2, 1	7	No

The method call `Hailstone.propLong(10)` returns `0.5`, since 5 of the 10 hailstone sequences shown in the table are considered long.

Write the `propLong` method. Assume that `hailstoneLength` and `isLongSeq` work as intended, regardless of what you wrote in parts (a) and (b). You must use `isLongSeq` appropriately to receive full credit.

```
/** Returns the proportion of the first n hailstone sequences that are considered long,
 *  as described in part (c).
 *  Precondition: n > 0
 */
public static double propLong(int n)
```

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 Hailstone

public static int hailstoneLength(int n)
public static boolean isLongSeq(int n)
public static double propLong(int n)
```

2. 2022.1

2. 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.

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()
```

(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)
```

Class information for this question

```
public class Level

public boolean goalReached()
public int getPoints()

public class Game

private Level levelOne
private Level levelTwo
private Level levelThree

public Game()
public boolean isBonus()
public void play()
public int getScore()
public int playManyTimes(int num)
```

3. 2019(2).1

3. 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.

}
```

2019(2).1

(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)
```

4. 2018.1

4. This question involves reasoning about a simulation of a frog hopping in a straight line. The frog attempts to hop to a goal within a specified number of hops. The simulation is encapsulated in the following `FrogSimulation` class. You will write two of the methods in this class.

```
public class FrogSimulation
{
   /** Distance, in inches, from the starting position to the goal. */
   private int goalDistance;

   /** Maximum number of hops allowed to reach the goal. */
   private int maxHops;


   /** Constructs a FrogSimulation where dist is the distance, in inches, from the starting
    *  position to the goal, and numHops is the maximum number of hops allowed to reach the goal.
    *  Precondition: dist > 0; numHops > 0
    */
   public FrogSimulation(int dist, int numHops)
   {
      goalDistance = dist;
      maxHops = numHops;
   }


   /** Returns an integer representing the distance, in inches, to be moved when the frog hops.
    */
   private int hopDistance()
   {   /* implementation not shown */   }


   /** Simulates a frog attempting to reach the goal as described in part (a).
    *  Returns true if the frog successfully reached or passed the goal during the simulation;
    *          false otherwise.
    */
   public boolean simulate()
   {   /* to be implemented in part (a) */   }


   /** Runs num simulations and returns the proportion of simulations in which the frog
    *  successfully reached or passed the goal.
    *  Precondition: num > 0
    */
   public double runSimulations(int num)
   {   /* to be implemented in part (b) */   }
}
```

(a) Write the `simulate` method, which simulates the frog attempting to hop in a straight line to a goal from the frog's starting position of 0 within a maximum number of hops. The method returns `true` if the frog successfully reached the goal within the maximum number of hops; otherwise, the method returns `false`.

The `FrogSimulation` class provides a method called `hopDistance` that returns an integer representing the distance (positive or negative) to be moved when the frog hops. A positive distance represents a move toward the goal. A negative distance represents a move away from the goal. The returned distance may vary from call to call. Each time the frog hops, its position is adjusted by the value returned by a call to the `hopDistance` method.

The frog hops until one of the following conditions becomes true:

- The frog has reached or passed the goal.
- The frog has reached a negative position.
- The frog has taken the maximum number of hops without reaching the goal.

The following example shows a declaration of a `FrogSimulation` object for which the goal distance is 24 inches and the maximum number of hops is 5. The table shows some possible outcomes of calling the `simulate` method.

```
FrogSimulation sim = new FrogSimulation(24, 5);
```

	Values returned by `hopDistance()`	Final position of frog	Return value of `sim.simulate()`
Example 1	`5, 7, -2, 8, 6`	`24`	`true`
Example 2	`6, 7, 6, 6`	`25`	`true`
Example 3	`6, -6, 31`	`31`	`true`
Example 4	`4, 2, -8`	`-2`	`false`
Example 5	`5, 4, 2, 4, 3`	`18`	`false`

Class information for this question

```
public class FrogSimulation

private int goalDistance
private int maxHops

private int hopDistance()
public boolean simulate()
public double runSimulations(int num)
```

Complete method `simulate` below. You must use `hopDistance` appropriately to receive full credit.

```
/** Simulates a frog attempting to reach the goal as described in part (a).
 *  Returns true if the frog successfully reached or passed the goal during the simulation;
 *          false otherwise.
 */
public boolean simulate()
```

(b) Write the `runSimulations` method, which performs a given number of simulations and returns the proportion of simulations in which the frog successfully reached or passed the goal. For example, if the parameter passed to `runSimulations` is `400`, and 100 of the 400 `simulate` method calls returned `true`, then the `runSimulations` method should return `0.25`.

Complete method `runSimulations` below. Assume that `simulate` works as specified, regardless of what you wrote in part (a). You must use `simulate` appropriately to receive full credit.

```
/** Runs num simulations and returns the proportion of simulations in which the frog
 *  successfully reached or passed the goal.
 *  Precondition: num > 0
 */
public double runSimulations(int num)
```

Array

1. 2005.4

1. Consider a grade-averaging scheme in which the final average of a student's scores is computed differently from the traditional average if the scores have "improved." Scores have improved if each score is greater than or equal to the previous score. The final average of the scores is computed as follows.

   A student has n scores indexed from 0 to n-1. If the scores have improved, only those scores with indexes greater than or equal to n/2 are averaged. If the scores have not improved, all the scores are averaged.

   The following table shows several lists of scores and how they would be averaged using the scheme described above.

Student Scores	Improved?	Final Average
50, 50, 20, 80, 53	No	(50 + 50 + 20 + 80 + 53) / 5.0 = 50.6
20, 50, 50, 53, 80	Yes	(50 + 53 + 80) / 3.0 = 61.0
20, 50, 50, 80	Yes	(50 + 80) / 2.0 = 65.0

   Consider the following incomplete StudentRecord class declaration. Each StudentRecord object stores a list of that student's scores and contains methods to compute that student's final average.

```
public class StudentRecord
{
  private int[] scores; // contains scores.length values
                        // scores.length > 1

  // constructors and other data fields not shown


  // returns the average (arithmetic mean) of the values in scores
  // whose subscripts are between first and last, inclusive
  // precondition:  0 <= first <= last < scores.length
  private double average(int first, int last)
  {  /* to be implemented in part (a) */  }


  // returns true if each successive value in scores is greater
  // than or equal to the previous value;
  // otherwise, returns false
  private boolean hasImproved()
  {  /* to be implemented in part (b) */  }


  // if the values in scores have improved, returns the average
  // of the elements in scores with indexes greater than or equal
  // to scores.length/2;
  // otherwise, returns the average of all of the values in scores
  public double finalAverage()
  {  /* to be implemented in part (c) */  }

}
```

(a) Write the StudentRecord method average. This method returns the average of the values in scores given a starting and an ending index.

Complete method average below.

```
// returns the average (arithmetic mean) of the values in scores
// whose subscripts are between first and last, inclusive
// precondition:  0 <= first <= last < scores.length
private double average(int first, int last)
```

(b) Write the StudentRecord method hasImproved.

Complete method hasImproved below.

```
// returns true if each successive value in scores is greater
// than or equal to the previous value;
// otherwise, returns false
private boolean hasImproved()
```

(c) Write the StudentRecord method finalAverage.

In writing finalAverage, you must call the methods defined in parts (a) and (b). Assume that these methods work as specified, regardless of what you wrote in parts (a) and (b).

Complete method finalAverage below.

```
// if the values in scores have improved, returns the average
// of the elements in scores with indexes greater than or equal
// to scores.length/2;
// otherwise, returns the average of all of the values in scores
public double finalAverage()
```

2. 2012.4

2. A grayscale image is represented by a 2-dimensional rectangular array of pixels (picture elements). A pixel is an integer value that represents a shade of gray. In this question, pixel values can be in the range from 0 through 255, inclusive. A black pixel is represented by 0, and a white pixel is represented by 255.

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

```
public class GrayImage
{
  public static final int BLACK = 0;
  public static final int WHITE = 255;


  /** The 2-dimensional representation of this image. Guaranteed not to be null.
   *  All values in the array are within the range [BLACK, WHITE], inclusive.
   */
  private int[][] pixelValues;


  /** @return the total number of white pixels in this image.
   *  Postcondition: this image has not been changed.
   */
  public int countWhitePixels()
  {  /* to be implemented in part (a) */  }


  /** Processes this image in row-major order and decreases the value of each pixel at
   *  position (row, col) by the value of the pixel at position (row + 2, col + 2) if it exists.
   *  Resulting values that would be less than BLACK are replaced by BLACK.
   *  Pixels for which there is no pixel at position (row + 2, col + 2) are unchanged.
   */
  public void processImage()
  {  /* to be implemented in part (b) */  }
}
```

(a) Write the method `countWhitePixels` that returns the number of pixels in the image that contain the value `WHITE`. For example, assume that `pixelValues` contains the following image.

	0	1	2	3	4
0	**255**	184	178	84	129
1	84	**255**	**255**	130	84
2	78	**255**	0	0	78
3	84	130	**255**	130	84

A call to `countWhitePixels` method would return 5 because there are 5 entries (shown in boldface) that have the value `WHITE`.

Complete method `countWhitePixels` below.

```
/** @return the total number of white pixels in this image.
 *  Postcondition: this image has not been changed.
 */
public int countWhitePixels()
```

(b) Write the method `processImage` that modifies the image by changing the values in the instance variable `pixelValues` according to the following description. The pixels in the image are processed one at a time in row-major order. Row-major order processes the first row in the array from left to right and then processes the second row from left to right, continuing until all rows are processed from left to right. The first index of `pixelValues` represents the row number, and the second index represents the column number.

The pixel value at position (row, col) is decreased by the value at position (row + 2, col + 2) if such a position exists. If the result of the subtraction is less than the value `BLACK`, the pixel is assigned the value of `BLACK`. The values of the pixels for which there is no pixel at position (row + 2, col + 2) remain unchanged. You may assume that all the original values in the array are within the range [`BLACK`, `WHITE`], inclusive.

The following diagram shows the contents of the instance variable `pixelValues` before and after a call to `processImage`. The values shown in boldface represent the pixels that could be modified in a grayscale image with 4 rows and 5 columns.

Before Call to `processImage`

	0	1	2	3	4
0	**221**	**184**	**178**	84	135
1	**84**	**255**	**255**	130	84
2	78	255	0	0	78
3	84	130	255	130	84

After Call to `processImage`

	0	1	2	3	4
0	**221**	**184**	**100**	84	135
1	**0**	**125**	**171**	130	84
2	78	255	0	0	78
3	84	130	255	130	84

Information repeated from the beginning of the question

```
public class GrayImage

public static final int BLACK = 0
public static final int WHITE = 255
private int[][] pixelValues
public int countWhitePixels()
public void processImage()
```

Complete method `processImage` below.

```
/** Processes this image in row-major order and decreases the value of each pixel at
 *  position (row, col) by the value of the pixel at position (row + 2, col + 2) if it exists.
 *  Resulting values that would be less than BLACK are replaced by BLACK.
 *  Pixels for which there is no pixel at position (row + 2, col + 2) are unchanged.
 */
public void processImage()
```

3. 2015.1

- 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.

3. This question involves reasoning about one-dimensional and two-dimensional arrays of integers. You will write three static methods, all of which are in a single enclosing class, named `DiverseArray` (not shown). The first method returns the sum of the values of a one-dimensional array; the second method returns an array that represents the sums of the rows of a two-dimensional array; and the third method analyzes row sums.

(a) Write a `static` method `arraySum` that calculates and returns the sum of the entries in a specified one-dimensional array. The following example shows an array `arr1` and the value returned by a call to `arraySum`.

		arr1			Value returned by arraySum(arr1)
0	1	2	3	4	
1	3	2	7	3	16

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Complete method `arraySum` below.

```
/** Returns the sum of the entries in the one-dimensional array arr.
 */
public static int arraySum(int[] arr)
```

(b) Write a `static` method `rowSums` that calculates the sums of each of the rows in a given two-dimensional array and returns these sums in a one-dimensional array. The method has one parameter, a two-dimensional array `arr2D` of `int` values. The array is in row-major order: `arr2D[r][c]` is the entry at row `r` and column `c`. The method returns a one-dimensional array with one entry for each row of `arr2D` such that each entry is the sum of the corresponding row in `arr2D`. As a reminder, each row of a two-dimensional array is a one-dimensional array.

For example, if `mat1` is the array represented by the following table, the call `rowSums(mat1)` returns the array `{16, 32, 28, 20}`.

mat1

	0	1	2	3	4
0	1	3	2	7	3
1	10	10	4	6	2
2	5	3	5	9	6
3	7	6	4	2	1

Methods written in this question

```
public static int arraySum(int[] arr)
public static int[] rowSums(int[][] arr2D)
public static boolean isDiverse(int[][] arr2D)
```

WRITE YOUR SOLUTION ON THE NEXT PAGE.

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

Complete method `rowSums` below.

```
/** Returns a one-dimensional array in which the entry at index k is the sum of
 *  the entries of row k of the two-dimensional array arr2D.
 */
public static int[] rowSums(int[][] arr2D)
```

(c) A two-dimensional array is *diverse* if no two of its rows have entries that sum to the same value. In the following examples, the array `mat1` is diverse because each row sum is different, but the array `mat2` is not diverse because the first and last rows have the same sum.

mat1

	0	1	2	3	4	Row sums
0	1	3	2	7	3	16
1	10	10	4	6	2	32
2	5	3	5	9	6	28
3	7	6	4	2	1	20

mat2

	0	1	2	3	4	Row sums
0	1	1	5	3	4	14
1	12	7	6	1	9	35
2	8	11	10	2	5	36
3	3	2	3	0	6	14

Write a `static` method `isDiverse` that determines whether or not a given two-dimensional array is diverse. The method has one parameter: a two-dimensional array `arr2D` of `int` values. The method should return `true` if all the row sums in the given array are unique; otherwise, it should return `false`. In the arrays shown above, the call `isDiverse(mat1)` returns `true` and the call `isDiverse(mat2)` returns `false`.

Methods written in this question

```
public static int arraySum(int[] arr)
public static int[] rowSums(int[][] arr2D)
public static boolean isDiverse(int[][] arr2D)
```

Assume that `arraySum` and `rowSums` work as specified, regardless of what you wrote in parts (a) and (b). You must use `rowSums` appropriately to receive full credit.

Complete method `isDiverse` below.

```
/** Returns true  if all rows in arr2D have different row sums;
 *          false otherwise.
 */
public static boolean isDiverse(int[][] arr2D)
```

4. 2018.4

4. This question involves reasoning about arrays of integers. You will write two static methods, both of which are in a class named ArrayTester.

```
public class ArrayTester
{

    /** Returns an array containing the elements of column c of arr2D in the same order as
     *  they appear in arr2D.
     *  Precondition: c is a valid column index in arr2D.
     *  Postcondition: arr2D is unchanged.
     */
    public static int[] getColumn(int[][] arr2D, int c)
    {  /* to be implemented in part (a) */  }


    /** Returns true if and only if every value in arr1 appears in arr2.
     *  Precondition: arr1 and arr2 have the same length.
     *  Postcondition: arr1 and arr2 are unchanged.
     */
    public static boolean hasAllValues(int[] arr1, int[] arr2)
    {  /* implementation not shown */  }


    /** Returns true if arr contains any duplicate values;
     *          false otherwise.
     */
    public static boolean containsDuplicates(int[] arr)
    {  /* implementation not shown */  }


    /** Returns true if square is a Latin square as described in part (b);
     *          false otherwise.
     *  Precondition: square has an equal number of rows and columns.
     *                square has at least one row.
     */
    public static boolean isLatin(int[][] square)
    {  /* to be implemented in part (b) */  }

}
```

(a) Write a static method `getColumn`, which returns a one-dimensional array containing the elements of a single column in a two-dimensional array. The elements in the returned array should be in the same order as they appear in the given column. The notation `arr2D[r][c]` represents the array element at row `r` and column `c`.

The following code segment initializes an array and calls the `getColumn` method.

```
int[][] arr2D = { { 0, 1, 2 },
                  { 3, 4, 5 },
                  { 6, 7, 8 },
                  { 9, 5, 3 } };

int[] result = ArrayTester.getColumn(arr2D, 1);
```

When the code segment has completed execution, the variable `result` will have the following contents.

```
result: {1, 4, 7, 5}
```

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Complete method `getColumn` below.

```
/** Returns an array containing the elements of column c of arr2D in the same order as they
 *  appear in arr2D.
 *  Precondition:  c is a valid column index in arr2D.
 *  Postcondition: arr2D is unchanged.
 */
public static int[] getColumn(int[][] arr2D, int c)
```

(b) Write the static method `isLatin,` which returns `true` if a given two-dimensional square array is a *Latin square*, and otherwise, returns `false`.

A two-dimensional square array of integers is a Latin square if the following conditions are true.

- The first row has no duplicate values.
- All values in the first row of the square appear in each row of the square.
- All values in the first row of the square appear in each column of the square.

Examples of Latin Squares

1	2	3
2	3	1
3	1	2

10	30	20	0
0	20	30	10
30	0	10	20
20	10	0	30

Examples that are NOT Latin Squares

1	2	1
2	1	1
1	1	2

Not a Latin square because the first row contains duplicate values

1	2	3
3	1	2
7	8	9

Not a Latin square because the elements of the first row do not all appear in the third row

1	2
1	2

Not a Latin square because the elements of the first row do not all appear in either column

The `ArrayTester` class provides two helper methods: `containsDuplicates` and `hasAllValues`. The method `containsDuplicates` returns `true` if the given one-dimensional array `arr` contains any duplicate values and `false` otherwise. The method `hasAllValues` returns `true` if and only if every value in `arr1` appears in `arr2`. You do not need to write the code for these methods.

Class information for this question

```
public class ArrayTester

public static int[] getColumn(int[][] arr2D, int c)
public static boolean hasAllValues(int[] arr1, int[] arr2)
public static boolean containsDuplicates(int[] arr)
public static boolean isLatin(int[][] square)
```

Complete method `isLatin` below. Assume that `getColumn` works as specified, regardless of what you wrote in part (a). You must use `getColumn`, `hasAllValues`, and `containsDuplicates` appropriately to receive full credit.

```
/** Returns true if square is a Latin square as described in part (b);
 *          false otherwise.
 *  Precondition: square has an equal number of rows and columns.
 *                square has at least one row.
 */
public static boolean isLatin(int[][] square)
```

5. 2021.4

5. This question involves manipulating a two-dimensional array of integers. You will write two `static` methods of the `ArrayResizer` class, which is shown below.

```
public class ArrayResizer
{
   /** Returns true if and only if every value in row r of array2D is non-zero.
    *  Precondition: r is a valid row index in array2D.
    *  Postcondition: array2D is unchanged.
    */
   public static boolean isNonZeroRow(int[][] array2D, int r)
   {  /* to be implemented in part (a) */  }

   /** Returns the number of rows in array2D that contain all non-zero values.
    *  Postcondition: array2D is unchanged.
    */
   public static int numNonZeroRows(int[][] array2D)
   {  /* implementation not shown */  }

   /** Returns a new, possibly smaller, two-dimensional array that contains only rows
    *  from array2D with no zeros, as described in part (b).
    *  Precondition: array2D contains at least one column and at least one row with no zeros.
    *  Postcondition: array2D is unchanged.
    */
   public static int[][] resize(int[][] array2D)
   {  /* to be implemented in part (b) */  }
}
```

(a) Write the method isNonZeroRow, which returns true if and only if all elements in row r of a two-dimensional array array2D are not equal to zero.

For example, consider the following statement, which initializes a two-dimensional array.

```
int[][] arr = {{2, 1, 0},
               {1, 3, 2},
               {0, 0, 0},
               {4, 5, 6}};
```

Sample calls to isNonZeroRow are shown below.

Call to isNonZeroRow	**Value Returned**	**Explanation**
ArrayResizer.isNonZeroRow(arr, 0)	false	At least one value in row 0 is zero.
ArrayResizer.isNonZeroRow(arr, 1)	true	All values in row 1 are non-zero.
ArrayResizer.isNonZeroRow(arr, 2)	false	At least one value in row 2 is zero.
ArrayResizer.isNonZeroRow(arr, 3)	true	All values in row 3 are non-zero.

Complete the isNonZeroRow method.

```
/** Returns true if and only if every value in row r of array2D is non-zero.
 *  Precondition: r is a valid row index in array2D.
 *  Postcondition: array2D is unchanged.
 */
public static boolean isNonZeroRow(int[][] array2D, int r)
```

(b) Write the method `resize`, which returns a new two-dimensional array containing only rows from `array2D` with all non-zero values. The elements in the new array should appear in the same order as the order in which they appeared in the original array.

The following code segment initializes a two-dimensional array and calls the `resize` method.

```
int[][] arr = {{2, 1, 0},
               {1, 3, 2},
               {0, 0, 0},
               {4, 5, 6}};
int[][] smaller = ArrayResizer.resize(arr);
```

When the code segment completes, the following will be the contents of `smaller`.

```
{{1, 3, 2}, {4, 5, 6}}
```

A helper method, `numNonZeroRows`, has been provided for you. The method returns the number of rows in its two-dimensional array parameter that contain no zero values.

Complete the `resize` method. Assume that `isNonZeroRow` works as specified, regardless of what you wrote in part (a). You must use `numNonZeroRows` and `isNonZeroRow` appropriately to receive full credit.

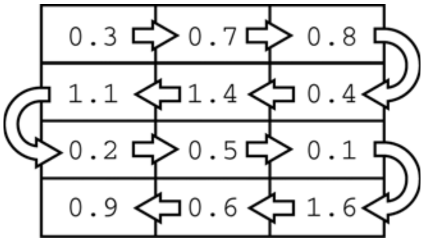
```
/** Returns a new, possibly smaller, two-dimensional array that contains only rows from array2D
 *  with no zeros, as described in part (b).
 *  Precondition: array2D contains at least one column and at least one row with no zeros.
 *  Postcondition: array2D is unchanged.
 */
public static int[][] resize(int[][] array2D)
```

Class information for this question

```
public class ArrayResizer

public static boolean isNonZeroRow(int[][] array2D, int r)
public static int numNonZeroRows(int[][] array2D)
public static int[][] resize(int[][] array2D)
```

6. 2013.4

6. A telescope scans a rectangular area of the night sky and collects the data into a 1-dimensional array. Each data value scanned is a number representing the amount of light detected by the telescope. The telescope scans back and forth across the sky (alternating between left to right and right to left) in the pattern indicated below by the arrows. The back-and-forth ordering of the values received from the scan is called *telescope order*.



0.3	0.7	0.8
1.1	1.4	0.4
0.2	0.5	0.1
0.9	0.6	1.6

The telescope records the data in telescope order into a 1-dimensional array of `double` values. This 1-dimensional array of information received from a single scan will be transferred into a 2-dimensional array, which reconstructs the original view of the rectangular area of the sky. This 2-dimensional array is part of the `SkyView` class, shown below. In this question you will write a constructor and a method for this class.

```
public class SkyView
{
   /** A rectangular array that holds the data representing a rectangular area of the sky. */
   private double[][] view;

   /** Constructs a SkyView object from a 1-dimensional array of scan data.
    *  @param numRows the number of rows represented in the view
    *          Precondition: numRows > 0
    *  @param numCols the number of columns represented in the view
    *          Precondition: numCols > 0
    *  @param scanned the scan data received from the telescope, stored in telescope order
    *          Precondition: scanned.length == numRows * numCols
    *  Postcondition: view has been created as a rectangular 2-dimensional array
    *                 with numRows rows and numCols columns and the values in
    *                 scanned have been copied to view and are ordered as
    *                 in the original rectangular area of sky.
    */
   public SkyView(int numRows, int numCols, double[] scanned)
   {   /* to be implemented in part (a) */   }

   /** Returns the average of the values in a rectangular section of view.
    *  @param startRow the first row index of the section
    *  @param endRow the last row index of the section
    *  @param startCol the first column index of the section
    *  @param endCol the last column index of the section
    *  Precondition: 0 <= startRow <= endRow < view.length
    *  Precondition: 0 <= startCol <= endCol < view[0].length
    *  @return the average of the values in the specified section of view
    */
   public double getAverage(int startRow, int endRow,
                            int startCol, int endCol)
   {   /* to be implemented in part (b) */   }

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

(a) Write the constructor for the `SkyView` class. The constructor initializes the `view` instance variable to a 2-dimensional array with `numRows` rows and `numCols` columns. The information from `scanned`, which is stored in the telescope order, is copied into `view` to reconstruct the sky view as originally seen by the telescope. The information in `scanned` must be rearranged as it is stored into `view` so that the sky view is oriented properly.

For example, suppose `scanned` contains values, as shown in the following array.

	0	1	2	3	4	5	6	7	8	9	10	11
scanned	0.3	0.7	0.8	0.4	1.4	1.1	0.2	0.5	0.1	1.6	0.6	0.9

Using the `scanned` array above, a `SkyView` object created with `new SkyView(4, 3, scanned)`, would have `view` initialized with the following values.

view	0	1	2
0	0.3	0.7	0.8
1	1.1	1.4	0.4
2	0.2	0.5	0.1
3	0.9	0.6	1.6

For another example, suppose `scanned` contains the following values.

	0	1	2	3	4	5
scanned	0.3	0.7	0.8	0.4	1.4	1.1

A `SkyView` object created with `new SkyView(3, 2, scanned)`, would have `view` initialized with the following values.

view	0	1
0	0.3	0.7
1	0.4	0.8
2	1.4	1.1

Complete the SkyView constructor below.

```
/** Constructs a SkyView object from a 1-dimensional array of scan data.
 *  @param numRows  the number of rows represented in the view
 *           Precondition: numRows > 0
 *  @param numCols  the number of columns represented in the view
 *           Precondition: numCols > 0
 *  @param scanned  the scan data received from the telescope, stored in telescope order
 *           Precondition: scanned.length == numRows * numCols
 *  Postcondition: view has been created as a rectangular 2-dimensional array
 *                  with numRows rows and numCols columns and the values in
 *                  scanned have been copied to view and are ordered as
 *                   in the original rectangular area of sky.
 */
public SkyView(int numRows, int numCols, double[] scanned)
```

(b) Write the `SkyView` method `getAverage`, which returns the average of the elements of the section of `view` with row indexes from `startRow` through `endRow`, inclusive, and column indexes from `startCol` through `endCol`, inclusive.

For example, if `nightSky` is a `SkyView` object where `view` contains the values shown below, the call `nightSky.getAverage(1, 2, 0, 1)` should return `0.8`. (The average is `(1.1 + 1.4 + 0.2 + 0.5) / 4`, which equals `0.8`). The section being averaged is indicated by the dark outline in the table below.

view	0	1	2
0	0.3	0.7	0.8
1	1.1	1.4	0.4
2	0.2	0.5	0.1
3	0.9	0.6	1.6

Class information repeated from the beginning of the question

```
public class SkyView

private double[][] view
public SkyView(int numRows, int numCols, double[] scanned)
public double getAverage(int startRow, int endRow,
                         int startCol, int endCol)
```

Complete method `getAverage` below.

```
/** Returns the average of the values in a rectangular section of view.
 *  @param startRow the first row index of the section
 *  @param endRow the last row index of the section
 *  @param startCol the first column index of the section
 *  @param endCol the last column index of the section
 *  Precondition: 0 <= startRow <= endRow < view.length
 *  Precondition: 0 <= startCol <= endCol < view[0].length
 *  @return the average of the values in the specified section of view
 */
public double getAverage(int startRow, int endRow,
                         int startCol, int endCol)
```

7. 2019(2).4

7. The `LightBoard` class models a two-dimensional display of lights, where each light is either on or off, as represented by a Boolean value. You will implement a constructor to initialize the display and a method to evaluate a light.

```
public class LightBoard
{
   /** The lights on the board, where true represents on and false represents off.
    */
   private boolean[][] lights;

   /** Constructs a LightBoard object having numRows rows and numCols columns.
    *  Precondition: numRows > 0, numCols > 0
    *  Postcondition: each light has a 40% probability of being set to on.
    */
   public LightBoard(int numRows, int numCols)
   {   /* to be implemented in part (a) */   }

   /** Evaluates a light in row index row and column index col and returns a status
    *  as described in part (b).
    *  Precondition: row and col are valid indexes in lights.
    */
   public boolean evaluateLight(int row, int col)
   {   /* to be implemented in part (b) */   }

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

}
```

(a) Write the constructor for the `LightBoard` class, which initializes `lights` so that each light is set to on with a 40% probability. The notation `lights[r][c]` represents the array element at row `r` and column `c`.

Complete the `LightBoard` constructor below.

```
/** Constructs a LightBoard object having numRows rows and numCols columns.
 *  Precondition: numRows > 0, numCols > 0
 *  Postcondition: each light has a 40% probability of being set to on.
 */
public LightBoard(int numRows, int numCols)
```

(b) Write the method `evaluateLight`, which computes and returns the status of a light at a given row and column based on the following rules.

1. If the light is on, return `false` if the number of lights in its column that are on is even, including the current light.
2. If the light is off, return `true` if the number of lights in its column that are on is divisible by three.
3. Otherwise, return the light's current status.

For example, suppose that `LightBoard sim = new LightBoard(7, 5)` creates a light board with the initial state shown below, where `true` represents a light that is on and `false` represents a light that is off. Lights that are off are shaded.

lights

	0	1	2	3	4
0	true	true	false	true	true
1	true	false	false	true	false
2	true	false	false	true	true
3	true	false	false	false	true
4	true	false	false	false	true
5	true	true	false	true	true
6	false	false	false	false	false

Sample calls to `evaluateLight` are shown below.

Call to `evaluateLight`	**Value Returned**	**Explanation**
`sim.evaluateLight(0, 3);`	`false`	The light is on, and the number of lights that are on in its column is even.
`sim.evaluateLight(6, 0);`	`true`	The light is off, and the number of lights that are on in its column is divisible by 3.
`sim.evaluateLight(4, 1);`	`false`	Returns the light's current status.
`sim.evaluateLight(5, 4);`	`true`	Returns the light's current status.

Class information for this question

```
public class LightBoard
private boolean[][] lights
public LightBoard(int numRows, int numCols)
public boolean evaluateLight(int row, int col)
```

Complete the `evaluateLight` method below.

```
/** Evaluates a light in row index row and column index col and returns a status
 *  as described in part (b).
 *  Precondition: row and col are valid indexes in lights.
 */
public boolean evaluateLight(int row, int col)
```

8. 2017.4

8. This question involves reasoning about a two-dimensional (2D) array of integers. You will write two static methods, both of which are in a single enclosing class named `Successors` (not shown). These methods process a 2D integer array that contains consecutive values. Each of these integers may be in any position in the 2D integer array. For example, the following 2D integer array with 3 rows and 4 columns contains the integers 5 through 16, inclusive.

2D Integer Array

	0	1	2	3
0	15	5	9	10
1	12	16	11	6
2	14	8	13	7

The following `Position` class is used to represent positions in the integer array. The notation `(r,c)` will be used to refer to a `Position` object with row `r` and column `c`.

```
public class Position
{
   /** Constructs a Position object with row r and column c. */
   public Position(int r, int c)
   {  /* implementation not shown */  }

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

(a) Write a `static` method `findPosition` that takes an integer value and a 2D integer array and returns the position of the integer in the given 2D integer array. If the integer is not an element of the 2D integer array, the method returns `null`.

For example, assume that array `arr` is the 2D integer array shown at the beginning of the question.

- The call `findPosition(8, arr)` would return the `Position` object `(2,1)` because the value `8` appears in `arr` at row 2 and column 1.
- The call `findPosition(17, arr)` would return `null` because the value `17` does not appear in `arr`.

Complete method `findPosition` below.

```
/** Returns the position of num in intArr;
 *  returns null if no such element exists in intArr.
 *  Precondition: intArr contains at least one row.
 */
public static Position findPosition(int num, int[][] intArr)
```

(b) Write a `static` method `getSuccessorArray` that returns a 2D successor array of positions created from a given 2D integer array.

The *successor* of an integer value is the integer that is one greater than that value. For example, the successor of 8 is 9. A 2D *successor array* shows the position of the successor of each element in a given 2D integer array. The 2D successor array has the same dimensions as the given 2D integer array. Each element in the 2D successor array is the position (row, column) of the corresponding 2D integer array element's successor. The largest element in the 2D integer array does not have a successor in the 2D integer array, so its corresponding position in the 2D successor array is `null`.

The following diagram shows a 2D integer array and its corresponding 2D successor array. To illustrate the successor relationship, the values 8 and 9 in the 2D integer array are shaded. In the 2D successor array, the shaded element shows that the position of the successor of 8 is `(0,2)` in the 2D integer array. The largest value in the 2D integer array is 16, so its corresponding element in the 2D successor array is `null`.

2D Integer Array

	0	1	2	3
0	15	5	9	10
1	12	16	11	6
2	14	8	13	7

2D Successor Array

	0	1	2	3
0	`(1,1)`	`(1,3)`	`(0,3)`	`(1,2)`
1	`(2,2)`	`null`	`(1,0)`	`(2,3)`
2	`(0,0)`	`(0,2)`	`(2,0)`	`(2,1)`

Class information for this question

```
public class Position

public Position(int r, int c)

public class Successors

public static Position findPosition(int num, int[][] intArr)
public static Position[][] getSuccessorArray(int[][] intArr)
```

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

Complete method `getSuccessorArray` below.

```
/** Returns a 2D successor array as described in part (b) constructed from intArr.
 *  Precondition: intArr contains at least one row and contains consecutive values.
 *                Each of these integers may be in any position in the 2D array.
 */
public static Position[][] getSuccessorArray(int[][] intArr)
```

9. 2019(1). 4

9. A theater contains rows of seats with the same number of seats in each row. Some rows contain tier 1 seats, and the remaining rows contain tier 2 seats. Tier 1 seats are closer to the stage and are more desirable. All seats in a row share the same tier.

The `Seat` class, shown below, represents seats in the theater. The `boolean` instance variable `available` is `false` if a ticket for the seat has been sold (the seat is no longer available). The `int` instance variable `tier` indicates whether the seat is a tier 1 or tier 2 seat.

```
public class Seat
{
   private boolean available;
   private int tier;

   public Seat(boolean isAvail, int tierNum)
   {
      available = isAvail;
      tier = tierNum;
   }

   public boolean isAvailable()
   {  return available;  }

   public int getTier()
   {  return tier;  }

   public void setAvailability(boolean isAvail)
   {  available = isAvail;  }
}
```

The Theater class represents a theater of seats. The number of seats per row and the number of tier 1 and tier 2 rows are determined by the parameters of the Theater constructor. Row 0 of the theaterSeats array represents the row closest to the stage.

```
public class Theater
{
   private Seat[][] theaterSeats;

   /** Constructs a Theater object, as described in part (a).
    *  Precondition: seatsPerRow > 0; tier1Rows > 0; tier2Rows >= 0
    */
   public Theater(int seatsPerRow, int tier1Rows, int tier2Rows)
   {  /* to be implemented in part (a) */  }

   /** Returns true if a seat holder was reassigned from the seat at fromRow, fromCol
    *  to the seat at toRow, toCol; otherwise it returns false, as described in part (b).
    *  Precondition: fromRow, fromCol, toRow, and toCol represent valid row and
    *                column positions in the theater.
    *                The seat at fromRow, fromCol is not available.
    */
   public boolean reassignSeat(int fromRow, int fromCol,
                               int toRow, int toCol)
   {  /* to be implemented in part (b) */  }
}
```

(a) Write the constructor for the `Theater` class. The constructor takes three `int` parameters, representing the number of seats per row, the number of tier 1 rows, and the number of tier 2 rows, respectively. The constructor initializes the `theaterSeats` instance variable so that it has the given number of seats per row and the given number of tier 1 and tier 2 rows and all seats are available and have the appropriate tier designation.

Row `0` of the `theaterSeats` array represents the row closest to the stage. All tier 1 seats are closer to the stage than tier 2 seats.

Complete the `Theater` constructor.

```
/** Constructs a Theater object, as described in part (a).
 *  Precondition: seatsPerRow > 0; tier1Rows > 0; tier2Rows >= 0
 */
public Theater(int seatsPerRow, int tier1Rows, int tier2Rows)
```

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 Seat

private boolean available
private int tier

public Seat(boolean isAvail, int tierNum)
public boolean isAvailable()
public int getTier()
public void setAvailability(boolean isAvail)

public class Theater

private Seat[][] theaterSeats

public Theater(int seatsPerRow, int tier1Rows, int tier2Rows)
public boolean reassignSeat(int fromRow, int fromCol,
                            int toRow, int toCol)
```

(b) Write the `reassignSeat` method, which attempts to move a person from a source seat to a destination seat. The reassignment can be made if the destination seat is available and has the same or greater tier than the source seat (that is, it is equally or less desirable). For example, a person in a tier 1 seat can be moved to a different tier 1 seat or to a tier 2 seat, but a person in a tier 2 seat can only be moved to a different tier 2 seat.

The `reassignSeat` method has four `int` parameters representing the row and column indexes of the source ("from") and destination ("to") seats. If the reassignment is possible, the source seat becomes available, the destination seat becomes unavailable, and the method returns `true`. If the seat reassignment is not possible, no changes are made to either seat and the method returns `false`. Assume that the source seat is occupied when the method is called.

Complete method `reassignSeat`.

```
/** Returns true if a seat holder was reassigned from the seat at fromRow, fromCol
 *  to the seat at toRow, toCol; otherwise it returns false, as described in part (b).
 *  Precondition: fromRow, fromCol, toRow, and toCol represent valid row and
 *                column positions in the theater.
 *                The seat at fromRow, fromCol is not available.
 */
public boolean reassignSeat(int fromRow, int fromCol,
                            int toRow, int toCol)
```

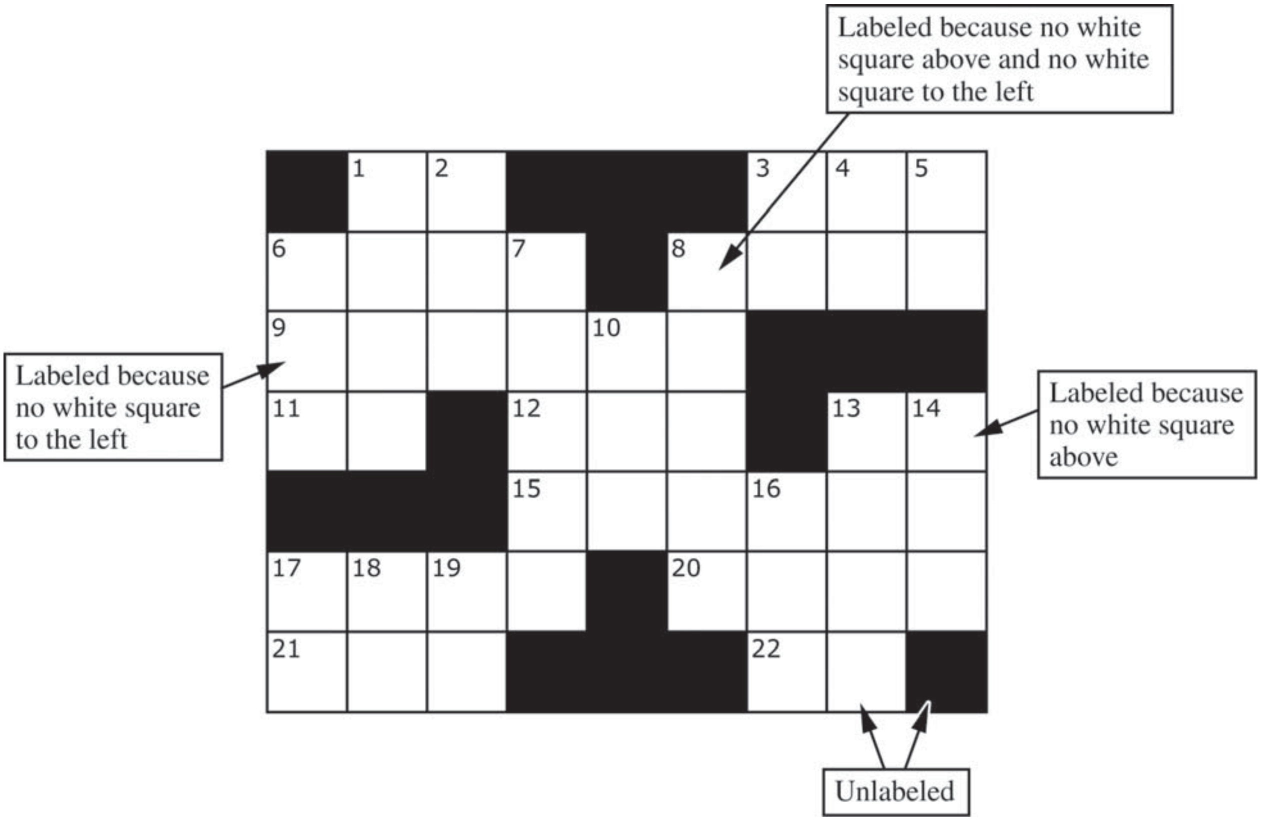
10. 2016.3

10. A crossword puzzle grid is a two-dimensional rectangular array of black and white squares. Some of the white squares are labeled with a positive number according to the *crossword labeling rule*.

The crossword labeling rule identifies squares to be labeled with a positive number as follows.

A square is labeled with a positive number if and only if
•! the square is white and
•! the square does not have a white square immediately above it, or it does not have a white square immediately to its left, or both.
The squares identified by these criteria are labeled with consecutive numbers in row-major order, starting at 1.

The following diagram shows a crossword puzzle grid and the labeling of the squares according to the crossword labeling rule.



Labeled because no white square above and no white square to the left
Labeled because no white square to the left
Labeled because no white square above
Unlabeled
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22

This question uses two classes, a `Square` class that represents an individual square in the puzzle and a `Crossword` class that represents a crossword puzzle grid. A partial declaration of the `Square` class is shown below.

```
public class Square
{
    /** Constructs one square of a crossword puzzle grid.
     *  Postcondition:
     *    – The square is black if and only if isBlack is true.
     *    – The square has number num.
     */
    public Square(boolean isBlack, int num)
    {  /* implementation not shown */  }

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

A partial declaration of the `Crossword` class is shown below. You will implement one method and the constructor in the `Crossword` class.

```
public class Crossword
{
    /** Each element is a Square object with a color (black or white) and a number.
     *  puzzle[r][c] represents the square in row r, column c.
     *  There is at least one row in the puzzle.
     */
    private Square[][] puzzle;

    /** Constructs a crossword puzzle grid.
     *  Precondition: There is at least one row in blackSquares.
     *  Postcondition:
     *    – The crossword puzzle grid has the same dimensions as blackSquares.
     *    – The Square object at row r, column c in the crossword puzzle grid is black
     *      if and only if blackSquares[r][c] is true.
     *    – The squares in the puzzle are labeled according to the crossword labeling rule.
     */
    public Crossword(boolean[][] blackSquares)
    {  /* to be implemented in part (b) */  }

    /** Returns true if the square at row r, column c should be labeled with a positive number;
     *          false otherwise.
     *  The square at row r, column c is black if and only if blackSquares[r][c] is true.
     *  Precondition: r and c are valid indexes in blackSquares.
     */
    private boolean toBeLabeled(int r, int c, boolean[][] blackSquares)
    {  /* to be implemented in part (a) */  }

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

Part (a) begins on page 14.

(a) Write the `Crossword` method `toBeLabeled`. The method returns `true` if the square indexed by row `r`, column `c` in a crossword puzzle grid should be labeled with a positive number according to the crossword labeling rule; otherwise it returns `false`. The parameter `blackSquares` indicates which squares in the crossword puzzle grid are black.

Class information for this question

```
public class Square

public Square(boolean isBlack, int num)

public class Crossword

private Square[][] puzzle

public Crossword(boolean[][] blackSquares)
private boolean toBeLabeled(int r, int c, boolean[][] blackSquares)
```

Complete method toBeLabeled below.

```
/** Returns true if the square at row r, column c should be labeled with a positive number;
 *         false otherwise.
 *  The square at row r, column c is black if and only if blackSquares[r][c] is true.
 *  Precondition: r and c are valid indexes in blackSquares.
 */
private boolean toBeLabeled(int r, int c, boolean[][] blackSquares)
```

(b) Write the `Crossword` constructor. The constructor should initialize the crossword puzzle grid to have the same dimensions as the parameter `blackSquares`. Each element of the puzzle grid should be initialized with a reference to a `Square` object with the appropriate color and number. The number is positive if the square is labeled and 0 if the square is not labeled.

Class information for this question

```
public class Square

public Square(boolean isBlack, int num)

public class Crossword

private Square[][] puzzle

public Crossword(boolean[][] blackSquares)
private boolean toBeLabeled(int r, int c, boolean[][] blackSquares)
```

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

Complete the Crossword constructor below.

```
/** Constructs a crossword puzzle grid.
 *  Precondition: There is at least one row in blackSquares.
 *  Postcondition:
 *    – The crossword puzzle grid has the same dimensions as blackSquares.
 *    – The Square object at row r, column c in the crossword puzzle grid is black
 *       if and only if blackSquares[r][c] is true.
 *    – The squares in the puzzle are labeled according to the crossword labeling rule.
 */
public Crossword(boolean[][] blackSquares)
```

11. 2009.3

11. An electric car that runs on batteries must be periodically recharged for a certain number of hours. The battery technology in the car requires that the charge time not be interrupted.

The cost for charging is based on the hour(s) during which the charging occurs. A rate table lists the 24 one-hour periods, numbered from 0 to 23, and the corresponding hourly cost for each period. The same rate table is used for each day. Each hourly cost is a positive integer. A sample rate table is given below.

Hour	**Cost**	**Hour**	**Cost**	**Hour**	**Cost**
0	50	8	150	16	200
1	60	9	150	17	200
2	160	10	150	18	180
3	60	11	200	19	180
4	80	12	40	20	140
5	100	13	240	21	100
6	100	14	220	22	80
7	120	15	220	23	60

The class BatteryCharger below uses a rate table to determine the most economic time to charge the battery. You will write two of the methods for the BatteryCharger class.

```
public class BatteryCharger
{
  /** rateTable has 24 entries representing the charging costs for hours 0 through 23. */
  private int[] rateTable;

  /** Determines the total cost to charge the battery starting at the beginning of startHour.
   *  @param startHour the hour at which the charge period begins
   *           Precondition: 0 ≤ startHour ≤ 23
   *  @param chargeTime the number of hours the battery needs to be charged
   *           Precondition: chargeTime > 0
   *  @return the total cost to charge the battery
   */
  private int getChargingCost(int startHour, int chargeTime)
  {  /* to be implemented in part (a) */  }

  /** Determines start time to charge the battery at the lowest cost for the given charge time.
   *  @param chargeTime the number of hours the battery needs to be charged
   *           Precondition: chargeTime > 0
   *  @return an optimal start time, with 0 ≤ returned value ≤ 23
   */
  public int getChargeStartTime(int chargeTime)
  {  /* to be implemented in part (b) */  }

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

(a) Write the `BatteryCharger` method `getChargingCost` that returns the total cost to charge a battery given the hour at which the charging process will start and the number of hours the battery needs to be charged.

For example, using the rate table given at the beginning of the question, the following table shows the resulting costs of several possible charges.

Start Hour of Charge	Hours of Charge Time	Last Hour of Charge	Total Cost
12	1	12	40
0	2	1	110
22	7	4 (the next day)	550
22	30	3 (two days later)	3,710

Note that a charge period consists of consecutive hours that may extend over more than one day.

Complete method `getChargingCost` below.

```
/** Determines the total cost to charge the battery starting at the beginning of startHour.
 *  @param startHour the hour at which the charge period begins
 *          Precondition: 0 ≤ startHour ≤ 23
 *  @param chargeTime the number of hours the battery needs to be charged
 *          Precondition: chargeTime > 0
 *  @return the total cost to charge the battery
 */
private int getChargingCost(int startHour, int chargeTime)
```

(b) Write the `BatteryCharger` method `getChargeStartTime` that returns the start time that will allow the battery to be charged at minimal cost. If there is more than one possible start time that produces the minimal cost, any of those start times can be returned.

For example, using the rate table given at the beginning of the question, the following table shows the resulting minimal costs and optimal starting hour of several possible charges.

Hours of Charge Time	Minimum Cost	Start Hour of Charge	Last Hour of Charge
1	40	12	12
2	110	0 **or** 23	1 0 (the next day)
7	550	22	4 (the next day)
30	3,710	22	3 (two days later)

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

Complete method `getChargeStartTime` below.

```
/** Determines start time to charge the battery at the lowest cost for the given charge time.
 *  @param chargeTime  the number of hours the battery needs to be charged
 *          Precondition: chargeTime > 0
 *  @return  an optimal start time, with 0 ≤ returned value ≤ 23
 */
public int getChargeStartTime(int chargeTime)
```

12. 2014.3

12. A student in a school is represented by the following class.

```
public class Student
{
   /** Returns the name of this Student. */
   public String getName()
   {  /* implementation not shown */  }

   /** Returns the number of times this Student has missed class. */
   public int getAbsenceCount()
   {  /* implementation not shown */  }

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

The class `SeatingChart`, shown below, uses a two-dimensional array to represent the seating arrangement of students in a classroom. The seats in the classroom are in a rectangular arrangement of rows and columns.

```
public class SeatingChart
{
   /** seats[r][c] represents the Student in row r and column c in the classroom. */
   private Student[][] seats;

   /** Creates a seating chart with the given number of rows and columns from the students in
    *  studentList. Empty seats in the seating chart are represented by null.
    *  @param rows the number of rows of seats in the classroom
    *  @param cols the number of columns of seats in the classroom
    *  Precondition: rows > 0; cols > 0;
    *                rows * cols >= studentList.size()
    *  Postcondition:
    *     - Students appear in the seating chart in the same order as they appear
    *       in studentList, starting at seats[0][0].
    *     - seats is filled column by column from studentList, followed by any
    *       empty seats (represented by null).
    *     - studentList is unchanged.
    */
   public SeatingChart(List<Student> studentList,
                       int rows, int cols)
   {  /* to be implemented in part (a) */  }

   /** Removes students who have more than a given number of absences from the
    *  seating chart, replacing those entries in the seating chart with null
    *  and returns the number of students removed.
    *  @param allowedAbsences an integer >= 0
    *  @return number of students removed from seats
    *  Postcondition:
    *    - All students with allowedAbsences or fewer are in their original positions in seats.
    *    - No student in seats has more than allowedAbsences absences.
    *    - Entries without students contain null.
    */
   public int removeAbsentStudents(int allowedAbsences)
   {  /* to be implemented in part (b) */  }

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

(a) Write the constructor for the `SeatingChart` class. The constructor initializes the `seats` instance variable to a two-dimensional array with the given number of rows and columns. The students in `studentList` are copied into the seating chart in the order in which they appear in `studentList`. The students are assigned to consecutive locations in the array `seats`, starting at `seats[0][0]` and filling the array column by column. Empty seats in the seating chart are represented by `null`.

For example, suppose a variable `List<Student> roster` contains references to `Student` objects in the following order.

"Karen" 3	"Liz" 1	"Paul" 4	"Lester" 1	"Henry" 5	"Renee" 9	"Glen" 2	"Fran" 6	"David" 1	"Danny" 3

A `SeatingChart` object created with the call `new SeatingChart(roster, 3, 4)` would have `seats` initialized with the following values.

	0	1	2	3
0	"Karen" 3	"Lester" 1	"Glen" 2	"Danny" 3
1	"Liz" 1	"Henry" 5	"Fran" 6	`null`
2	"Paul" 4	"Renee" 9	"David" 1	`null`

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Complete the `SeatingChart` constructor below.

```
/** Creates a seating chart with the given number of rows and columns from the students in
 *  studentList. Empty seats in the seating chart are represented by null.
 *  @param rows the number of rows of seats in the classroom
 *  @param cols the number of columns of seats in the classroom
 *  Precondition: rows > 0; cols > 0;
 *                rows * cols >= studentList.size()
 *  Postcondition:
 *      - Students appear in the seating chart in the same order as they appear
 *        in studentList, starting at seats[0][0].
 *      - seats is filled column by column from studentList, followed by any
 *        empty seats (represented by null).
 *      - studentList is unchanged.
 */
public SeatingChart(List<Student> studentList,
                    int rows, int cols)
```

(b) Write the `removeAbsentStudents` method, which removes students who have more than a given number of absences from the seating chart and returns the number of students that were removed. When a student is removed from the seating chart, a `null` is placed in the entry for that student in the array `seats`. For example, suppose the variable `SeatingChart introCS` has been created such that the array `seats` contains the following entries showing both students and their number of absences.

	0	1	2	3
0	"Karen" 3	"Lester" 1	"Glen" 2	"Danny" 3
1	"Liz" 1	"Henry" 5	"Fran" 6	`null`
2	"Paul" 4	"Renee" 9	"David" 1	`null`

After the call `introCS.removeAbsentStudents(4)` has executed, the array `seats` would contain the following values and the method would return the value 3.

	0	1	2	3
0	"Karen" 3	"Lester" 1	"Glen" 2	"Danny" 3
1	"Liz" 1	**`null`**	**`null`**	`null`
2	"Paul" 4	**`null`**	"David" 1	`null`

Class information repeated from the beginning of the question:

```
public class Student

public String getName()
public int getAbsenceCount()

public class SeatingChart

private Student[][] seats
public SeatingChart(List<Student> studentList,
                    int rows, int cols)
public int removeAbsentStudents(int allowedAbsences)
```

Complete method removeAbsentStudents below.

```
/** Removes students who have more than a given number of absences from the
 *  seating chart, replacing those entries in the seating chart with null
 *  and returns the number of students removed.
 *  @param allowedAbsences an integer >= 0
 *  @return number of students removed from seats
 *  Postcondition:
 *    - All students with allowedAbsences or fewer are in their original positions in seats.
 *    - No student in seats has more than allowedAbsences absences.
 *    - Entries without students contain null.
 */
public int removeAbsentStudents(int allowedAbsences)
```

13. 2013.2

16. A multiplayer game called Token Pass has the following rules.

Each player begins with a random number of tokens (at least 1, but no more than 10) that are placed on a linear game board. There is one position on the game board for each player. After the game board has been filled, a player is randomly chosen to begin the game. Each position on the board is numbered, starting with 0.

The following rules apply for a player’s turn.

- The tokens are collected and removed from the game board at that player's position.
- The collected tokens are distributed one at a time, to each player, beginning with the next player in order of increasing position.
- If there are still tokens to distribute after the player at the highest position gets a token, the next token will be distributed to the player at position 0.
- The distribution of tokens continues until there are no more tokens to distribute.

The Token Pass game board is represented by an array of integers. The indexes of the array represent the player positions on the game board, and the corresponding values in the array represent the number of tokens that each player has. The following example illustrates one player's turn.

<u>Example</u>
The following represents a game with 4 players. The player at position 2 was chosen to go first.

			↓	
Player	0	1	2	3
Tokens	3	2	6	10

The tokens at position 2 are collected and distributed as follows.

1st token - to position 3 (The highest position is reached, so the next token goes to position 0.)
2nd token - to position 0
3rd token - to position 1
4th token - to position 2
5th token - to position 3 (The highest position is reached, so the next token goes to position 0.)
6th token - to position 0

After player 2’s turn, the values in the array will be as follows.

			↓	
Player	0	1	2	3
Tokens	5	3	1	12

The Token Pass game is represented by the `TokenPass` class.

```
public class TokenPass
{
  private int[] board;
  private int currentPlayer;


  /** Creates the board array to be of size playerCount and fills it with
   *  random integer values from 1 to 10, inclusive. Initializes currentPlayer to a
   *  random integer value in the range between 0 and playerCount-1, inclusive.
   *  @param playerCount the number of players
   */
  public TokenPass(int playerCount)
  {   /* to be implemented in part (a) */   }



  /** Distributes the tokens from the current player's position one at a time to each player in
   *  the game. Distribution begins with the next position and continues until all the tokens
   *  have been distributed. If there are still tokens to distribute when the player at the
   *  highest position is reached, the next token will be distributed to the player at position 0.
   *  Precondition: the current player has at least one token.
   *  Postcondition: the current player has not changed.
   */
  public void distributeCurrentPlayerTokens()
  {   /* to be implemented in part (b) */   }

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

(a) Write the constructor for the `TokenPass` class. The parameter `playerCount` represents the number of players in the game. The constructor should create the `board` array to contain `playerCount` elements and fill the array with random numbers between 1 and 10, inclusive. The constructor should also initialize the instance variable `currentPlayer` to a random number between 0 and `playerCount-1`, inclusive.

Complete the `TokenPass` constructor below.

```
/** Creates the board array to be of size playerCount and fills it with
 *  random integer values from 1 to 10, inclusive. Initializes currentPlayer to a
 *  random integer value in the range between 0 and playerCount-1, inclusive.
 *  @param playerCount the number of players
 */
public TokenPass(int playerCount)
```

(b) Write the `distributeCurrentPlayerTokens` method.

The tokens are collected and removed from the game board at the current player's position. These tokens are distributed, one at a time, to each player, beginning with the next higher position, until there are no more tokens to distribute.

Class information repeated from the beginning of the question

```
public class TokenPass

private int[] board
private int currentPlayer
public TokenPass(int playerCount)
public void distributeCurrentPlayerTokens()
```

Complete method `distributeCurrentPlayerTokens` below.

```
/** Distributes the tokens from the current player's position one at a time to each player in
 *  the game. Distribution begins with the next position and continues until all the tokens
 *  have been distributed. If there are still tokens to distribute when the player at the
 *  highest position is reached, the next token will be distributed to the player at position 0.
 *  Precondition: the current player has at least one token.
 *  Postcondition: the current player has not changed.
 */
public void distributeCurrentPlayerTokens()
```

14. 2012.3

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

```
public class Horse  {

   /** @return  the horse's name  */
   String getName();

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

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

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

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

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


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


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

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

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

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

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

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

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

Information repeated from the beginning of the question

```
public class Horse

String getName()
int getWeight()

public class HorseBarn

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

Complete method `findHorseSpace` below.

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

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

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

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

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

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

Information repeated from the beginning of the question

```
public class Horse

String getName()
int getWeight()

public class HorseBarn

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

Complete method `consolidate` below.

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

15. 2007.1

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

18. A positive integer is called a "self-divisor" if every decimal digit of the number is a divisor of the number, that is, the number is evenly divisible by each and every one of its digits. For example, the number 128 is a self-divisor because it is evenly divisible by 1, 2, and 8. However, 26 is not a self-divisor because it is not evenly divisible by the digit 6. Note that 0 is not considered to be a divisor of any number, so any number containing a 0 digit is NOT a self-divisor. There are infinitely many self-divisors.

```
public class SelfDivisor
{
  /** @param number  the number to be tested
   *          Precondition:  number > 0
   *  @return true  if every decimal digit of  number  is a divisor of  number;
   *           false  otherwise
   */
  public static boolean isSelfDivisor(int number)
  {   /*  to be implemented in part (a)  */    }


  /** @param start  starting point for values to be checked
   *          Precondition:  start > 0
   *  @param num  the size of the array to be returned
   *          Precondition:  num > 0
   *  @return  an array containing the first  num  integers ≥ start  that are self-divisors
   */
  public static int[] firstNumSelfDivisors(int start, int num)
  {   /*  to be implemented in part (b)  */    }


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

(a) Write method isSelfDivisor, which takes a positive integer as its parameter. This method returns true if the number is a self-divisor; otherwise, it returns false.

Complete method isSelfDivisor below.

```
/** @param number  the number to be tested
 *         Precondition: number > 0
 *  @return true if every decimal digit of number is a divisor of number;
 *          false otherwise
 */
public static boolean isSelfDivisor(int number)
```

(b) Write method firstNumSelfDivisors, which takes two positive integers as parameters, representing a start value and a number of values. Method firstNumSelfDivisors returns an array of size num that contains the first num self-divisors that are greater than or equal to start.

For example, the call firstNumSelfDivisors(10, 3) should return an array containing the values 11, 12, and 15, because the first three self-divisors that are greater than or equal to 10 are 11, 12, and 15.

In writing firstNumSelfDivisors, assume that isSelfDivisor works as specified, regardless of what you wrote in part (a).

Complete method firstNumSelfDivisors below.

```
/** @param start  starting point for values to be checked
 *         Precondition: start > 0
 *  @param num  the size of the array to be returned
 *         Precondition: num > 0
 *  @return an array containing the first num integers ≥ start that are self-divisors
 */
public static int[] firstNumSelfDivisors(int start, int num)
```

16. 2011.1

16. Digital sounds can be represented as an array of integer values. For this question, you will write two unrelated methods of the Sound class.

A partial declaration of the Sound class is shown below.

```
public class Sound
{
   /** the array of values in this sound; guaranteed not to be null */
   private int[] samples;

   /** Changes those values in this sound that have an amplitude greater than limit.
    *  Values greater than limit are changed to limit.
    *  Values less than -limit are changed to -limit.
    *  @param limit the amplitude limit
    *           Precondition: limit ≥ 0
    *  @return the number of values in this sound that this method changed
    */
   public int limitAmplitude(int limit)
   {  /* to be implemented in part (a) */  }

   /** Removes all silence from the beginning of this sound.
    *  Silence is represented by a value of 0.
    *  Precondition: samples contains at least one nonzero value
    *  Postcondition: the length of samples reflects the removal of starting silence
    */
   public void trimSilenceFromBeginning()
   {  /* to be implemented in part (b) */  }

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

(a) The volume of a sound depends on the amplitude of each value in the sound. The amplitude of a value is its absolute value. For example, the amplitude of -2300 is 2300 and the amplitude of 4000 is 4000.

Write the method `limitAmplitude` that will change any value that has an amplitude greater than the given limit. Values that are greater than `limit` are replaced with `limit`, and values that are less than `-limit` are replaced with `-limit`. The method returns the total number of values that were changed in the array. For example, assume that the array `samples` has been initialized with the following values.

40	2532	17	-2300	-17	-4000	2000	1048	-420	33	15	-32	2030	3223

When the statement

```
int numChanges = limitAmplitude(2000);
```

is executed, the value of `numChanges` will be 5, and the array `samples` will contain the following values.

40	2000	17	-2000	-17	-2000	2000	1048	-420	33	15	-32	2000	2000

Complete method `limitAmplitude` below.

```
/** Changes those values in this sound that have an amplitude greater than limit.
 *  Values greater than limit are changed to limit.
 *  Values less than -limit are changed to -limit.
 *  @param limit the amplitude limit
 *           Precondition: limit ≥ 0
 *  @return the number of values in this sound that this method changed
 */
public int limitAmplitude(int limit)
```

(b) Recorded sound often begins with silence. Silence in a sound is represented by a value of 0.

Write the method `trimSilenceFromBeginning` that removes the silence from the beginning of a sound. To remove starting silence, a new array of values is created that contains the same values as the original `samples` array in the same order but without the leading zeros. The instance variable `samples` is updated to refer to the new array. For example, suppose the instance variable `samples` refers to the following array.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Value	0	0	0	0	-14	0	-35	-39	0	-7	16	32	37	29	0	0

After `trimSilenceFromBeginning` has been called, the instance variable `samples` will refer to the following array.

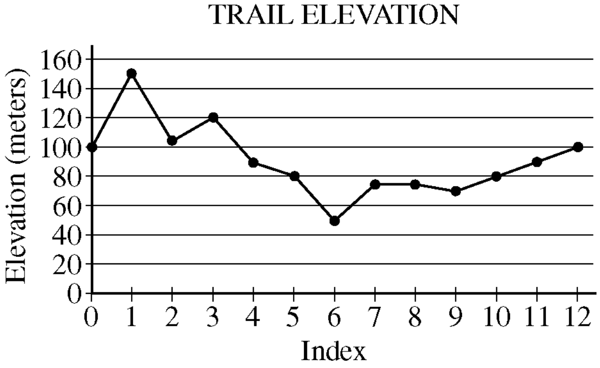
Index	0	1	2	3	4	5	6	7	8	9	10	11
Value	-14	0	-35	-39	0	-7	16	32	37	29	0	0

Complete method `trimSilenceFromBeginning` below.

```
/** Removes all silence from the beginning of this sound.
 *  Silence is represented by a value of 0.
 *  Precondition: samples contains at least one nonzero value
 *  Postcondition: the length of samples reflects the removal of starting silence
 */
public void trimSilenceFromBeginning()
```

17. 2010.3

17. A hiking trail has elevation markers posted at regular intervals along the trail. Elevation information about a trail can be stored in an array, where each element in the array represents the elevation at a marker. The elevation at the first marker will be stored at array index 0, the elevation at the second marker will be stored at array index 1, and so forth. Elevations between markers are ignored in this question. The graph below shows an example of trail elevations.



The table below contains the data represented in the graph.

Trail Elevation (meters)

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Elevation	100	150	105	120	90	80	50	75	75	70	80	90	100

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

```
public class Trail
{
   /** Representation of the trail. The number of markers on the trail is markers.length. */
   private int[] markers;

   /** Determines if a trail segment is level. A trail segment is defined by a starting marker,
    *   an ending marker, and all markers between those two markers.
    *   A trail segment is level if it has a difference between the maximum elevation
    *   and minimum elevation that is less than or equal to 10 meters.
    *   @param start the index of the starting marker
    *   @param end the index of the ending marker
    *           Precondition: 0 <= start < end <= markers.length - 1
    *   @return true if the difference between the maximum and minimum
    *                elevation on this segment of the trail is less than or equal to 10 meters;
    *                false otherwise.
    */
   public boolean isLevelTrailSegment(int start, int end)
   {  /* to be implemented in part (a) */  }

   /** Determines if this trail is rated difficult. A trail is rated by counting the number of changes in
    *   elevation that are at least 30 meters (up or down) between two consecutive markers. A trail
    *   with 3 or more such changes is rated difficult.
    *   @return true if the trail is rated difficult; false otherwise.
    */
   public boolean isDifficult()
   {  /* to be implemented in part (b) */  }

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

(a) Write the Trail method isLevelTrailSegment. A trail segment is defined by a starting marker, an ending marker, and all markers between those two markers. The parameters of the method are the index of the starting marker and the index of the ending marker. The method will return true if the difference between the maximum elevation and the minimum elevation in the trail segment is less than or equal to 10 meters.

For the trail shown at the beginning of the question, the trail segment starting at marker 7 and ending at marker 10 has elevations ranging between 70 and 80 meters. Because the difference between 80 and 70 is equal to 10, the trail segment is considered level.

The trail segment starting at marker 2 and ending at marker 12 has elevations ranging between 50 and 120 meters. Because the difference between 120 and 50 is greater than 10, this trail segment is not considered level.

Complete method `isLevelTrailSegment` below.

```
/** Determines if a trail segment is level. A trail segment is defined by a starting marker,
 *  an ending marker, and all markers between those two markers.
 *  A trail segment is level if it has a difference between the maximum elevation
 *  and minimum elevation that is less than or equal to 10 meters.
 *  @param start the index of the starting marker
 *  @param end the index of the ending marker
 *          Precondition: 0 <= start < end <= markers.length - 1
 *  @return true if the difference between the maximum and minimum
 *               elevation on this segment of the trail is less than or equal to 10 meters;
 *               false otherwise.
 */
public boolean isLevelTrailSegment(int start, int end)
```

(b) Write the `Trail` method `isDifficult`. A trail is rated by counting the number of changes in elevation that are at least 30 meters (up or down) between two consecutive markers. A trail with 3 or more such changes is rated difficult. The following table shows trail elevation data and the elevation changes between consecutive trail markers.

Trail Elevation (meters)

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Elevation	100	150	105	120	90	80	50	75	75	70	80	90	100

```
                   \ /   \ /   \ /   \ /   \ /   \ /   \ /   \ /   \ /   \ /   \ /   \ /
Elevation change   50    -45   15    -30   -10   -30   25    0     -5    10    10    10
```

This trail is rated difficult because it has 4 changes in elevation that are 30 meters or more (between markers 0 and 1, between markers 1 and 2, between markers 3 and 4, and between markers 5 and 6).

Complete method `isDifficult` below.

```
/** Determines if this trail is difficult. A trail is rated by counting the number of changes in
 *  elevation that are at least 30 meters (up or down) between two consecutive markers. A trail
 *  with 3 or more such changes is rated difficult.
 *  @return true if the trail is rated difficult; false otherwise.
 */
public boolean isDifficult()
```

18. 2009.1

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

21. A statistician is studying sequences of numbers obtained by repeatedly tossing a six-sided number cube. On each side of the number cube is a single number in the range of 1 to 6, inclusive, and no number is repeated on the cube. The statistician is particularly interested in runs of numbers. A run occurs when two or more consecutive tosses of the cube produce the same value. For example, in the following sequence of cube tosses, there are runs starting at positions 1, 6, 12, and 14.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Result	1	**5**	**5**	4	3	1	**2**	**2**	**2**	**2**	6	1	**3**	**3**	**5**	**5**	**5**	**5**

The number cube is represented by the following class.

```
public class NumberCube
{
   /** @return  an integer value between 1 and 6, inclusive
    */
   public int toss()
   {   /*  implementation not shown  */   }

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

You will implement a method that collects the results of several tosses of a number cube and another method that calculates the longest run found in a sequence of tosses.

(a) Write the method `getCubeTosses` that takes a number cube and a number of tosses as parameters. The method should return an array of the values produced by tossing the number cube the given number of times.

Complete method `getCubeTosses` below.

```
/** Returns an array of the values obtained by tossing a number cube numTosses times.
 *  @param cube a NumberCube
 *  @param numTosses the number of tosses to be recorded
 *          Precondition: numTosses > 0
 *  @return an array of numTosses values
 */
public static int[] getCubeTosses(NumberCube cube, int numTosses)
```

(b) Write the method `getLongestRun` that takes as its parameter an array of integer values representing a series of number cube tosses. The method returns the starting index in the array of a run of maximum size. A run is defined as the repeated occurrence of the same value in two or more consecutive positions in the array.

For example, the following array contains two runs of length 4, one starting at index 6 and another starting at index 14. The method may return either of those starting indexes.

If there are no runs of any value, the method returns `-1`.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Result	1	**5**	**5**	4	3	1	**2**	**2**	**2**	**2**	6	1	**3**	**3**	**5**	**5**	**5**	**5**

Complete method `getLongestRun` below.

```
/** Returns the starting index of a longest run of two or more consecutive repeated values
 *  in the array values.
 *  @param values an array of integer values representing a series of number cube tosses
 *          Precondition: values.length > 0
 *  @return the starting index of a run of maximum size;
 *           -1 if there is no run
 */
public static int getLongestRun(int[] values)
```

19. 2006.3

19. Consider the following incomplete class that stores information about a customer, which includes a name and unique ID (a positive integer). To facilitate sorting, customers are ordered alphabetically by name. If two or more customers have the same name, they are further ordered by ID number. A particular customer is "greater than" another customer if that particular customer appears later in the ordering than the other customer.

```
public class Customer
{
  // constructs a Customer with given name and ID number
  public Customer(String name, int idNum)
  {  /* implementation not shown */  }

  // returns the customer's name
  public String getName()
  {  /* implementation not shown */  }

  // returns the customer's id
  public int getID()
  {  /* implementation not shown */  }


  // returns 0 when this customer is equal to other;
  //   a positive integer when this customer is greater than other;
  //   a negative integer when this customer is less than other
  public int compareCustomer(Customer other)
  {  /* to be implemented in part (a) */  }

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

(a) Write the `Customer` method `compareCustomer`, which compares this customer to a given customer, `other`. Customers are ordered alphabetically by name, using the `compareTo` method of the `String` class. If the names of the two customers are the same, then the customers are ordered by ID number. Method `compareCustomer` should return a positive integer if this customer is greater than `other`, a negative integer if this customer is less than `other`, and 0 if they are the same.

For example, suppose we have the following `Customer` objects.

```
Customer c1 = new Customer("Smith", 1001);
Customer c2 = new Customer("Anderson", 1002);
Customer c3 = new Customer("Smith", 1003);
```

The following table shows the result of several calls to `compareCustomer`.

Method Call	Result
`c1.compareCustomer(c1)`	0
`c1.compareCustomer(c2)`	a positive integer
`c1.compareCustomer(c3)`	a negative integer

Complete method `compareCustomer` below.

```
// returns 0 when this customer is equal to other;
//   a positive integer when this customer is greater than other;
//   a negative integer when this customer is less than other
public int compareCustomer(Customer other)
```

(b) A company maintains customer lists where each list is a sorted array of customers stored in ascending order by customer. A customer may appear in more than one list, but will not appear more than once in the same list.

Write method `prefixMerge`, which takes three array parameters. The first two arrays, `list1` and `list2`, represent existing customer lists. It is possible that some customers are in both arrays. The third array, `result`, has been instantiated to a length that is no longer than either of the other two arrays and initially contains `null` values. Method `prefixMerge` uses an algorithm similar to the merge step of a Mergesort to fill the array `result`. Customers are copied into `result` from the beginning of `list1` and `list2`, merging them in ascending order until all positions of `result` have been filled. Customers who appear in both `list1` and `list2` will appear at most once in `result`.

For example, assume that three arrays have been initialized as shown below.

`list1`	Arthur 4920	Burton 3911	Burton 4944	Franz 1692	Horton 9221	Jones 5554	Miller 9360	Nguyen 4339
	`[0]`	`[1]`	`[2]`	`[3]`	`[4]`	`[5]`	`[6]`	`[7]`

`list2`	Aaron 1729	Baker 2921	Burton 3911	Dillard 6552	Jones 5554	Miller 9360	Noble 3335
	`[0]`	`[1]`	`[2]`	`[3]`	`[4]`	`[5]`	`[6]`

`result`	`null`	`null`	`null`	`null`	`null`	`null`
	`[0]`	`[1]`	`[2]`	`[3]`	`[4]`	`[5]`

In this example, the array `result` must contain the following values after the call `prefixMerge(list1, list2, result)`.

`result`	Aaron 1729	Arthur 4920	Baker 2921	Burton 3911	Burton 4944	Dillard 6552
	`[0]`	`[1]`	`[2]`	`[3]`	`[4]`	`[5]`

In writing `prefixMerge`, you may assume that `compareCustomer` works as specified, regardless of what you wrote in part (a). Solutions that create any additional data structures holding multiple objects (e.g., arrays, `ArrayLists`, etc.) will not receive full credit.

Complete method `prefixMerge` below.

```
// fills result with customers merged from the
// beginning of list1 and list2;
// result contains no duplicates and is sorted in
// ascending order by customer
// precondition:  result.length > 0;
//                list1.length >= result.length;
//                list1 contains no duplicates;
//                list2.length >= result.length;
//                list2 contains no duplicates;
//                list1 and list2 are sorted in
//                ascending order by customer
// postcondition: list1, list2 are not modified
public static void prefixMerge(Customer[] list1,
                               Customer[] list2,
                               Customer[] result)
```

20. 2005.1

20. In this question, you will implement two methods for a class `Hotel` that is part of a hotel reservation system. The `Hotel` class uses the `Reservation` class shown below. A `Reservation` is for the person and room number specified when the `Reservation` is constructed.

```
public class Reservation
{
  public Reservation(String guestName, int roomNumber)
  { /* implementation not shown */ }

  public int getRoomNumber()
  { /* implementation not shown */ }

  // private data and other methods not shown
}
```

An incomplete declaration for the `Hotel` class is shown below. Each hotel in the hotel reservation system has rooms numbered 0, 1, 2, . . . , up to the last room number in the hotel. For example, a hotel with 100 rooms would have rooms numbered 0, 1, 2, . . . , 99.

```
public class Hotel
{
  private Reservation[] rooms;
    // each element corresponds to a room in the hotel;
    // if rooms[index] is null, the room is empty;
    // otherwise, it contains a reference to the Reservation
    // for that room, such that
    // rooms[index].getRoomNumber() returns index

  private ArrayList waitList;
    // contains names of guests who have not yet been
    // assigned a room because all rooms are full


  // if there are any empty rooms (rooms with no reservation),
  // then create a reservation for an empty room for the
  // specified guest and return the new Reservation;
  // otherwise, add the guest to the end of waitList
  // and return null
  public Reservation requestRoom(String guestName)
  { /* to be implemented in part (a) */ }


  // release the room associated with parameter res, effectively
  // canceling the reservation;
  // if any names are stored in waitList, remove the first name
  // and create a Reservation for this person in the room
  // reserved by res; return that new Reservation;
  // if waitList is empty, mark the room specified by res as empty and
  // return null
  // precondition:  res is a valid Reservation for some room
  //                in this hotel
  public Reservation cancelAndReassign(Reservation res)
  { /* to be implemented in part (b) */ }

  // constructors and other methods not shown
}
```

(a) Write the Hotel method requestRoom. Method requestRoom attempts to reserve a room in the hotel for a given guest. If there are any empty rooms in the hotel, one of them will be assigned to the named guest and the newly created reservation is returned. If there are no empty rooms, the guest is added to the end of the waiting list and null is returned.

Complete method requestRoom below.

```
// if there are any empty rooms (rooms with no reservation),
// then create a reservation for an empty room for the
// specified guest and return the new Reservation;
// otherwise, add the guest to the end of waitList
// and return null
public Reservation requestRoom(String guestName)
```

(b) Write the Hotel method cancelAndReassign. Method cancelAndReassign releases a previous reservation. If the waiting list for the hotel contains any names, the vacated room is reassigned to the first person at the beginning of the list. That person is then removed from the waiting list and the newly created reservation is returned. If no one is waiting, the room is marked as empty and null is returned.

In writing cancelAndReassign you may call any accessible methods in the Reservation and Hotel classes. Assume that these methods work as specified.

Complete method cancelAndReassign below.

```
// release the room associated with parameter res, effectively
// canceling the reservation;
// if any names are stored in waitList, remove the first name
// and create a Reservation for this person in the room
// reserved by res; return that new Reservation;
// if waitList is empty, mark the room specified by res as empty and
// return null
// precondition:  res is a valid Reservation for some room
//                in this hotel
public Reservation cancelAndReassign(Reservation res)
```

21. 2022.4

21. 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.
}
```

(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()
```

(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()
```

Class information for this question

```
public class Data

public static final int MAX = /* value not shown */
private int[][] grid

public void repopulate()
public int countIncreasingCols()
```

String FRQ

1. 2021.1

1. This question involves the `WordMatch` class, which stores a secret string and provides methods that compare other strings to the secret string. You will write two methods in the `WordMatch` class.

```
public class WordMatch
{
   /** The secret string. */
   private String secret;

   /** Constructs a WordMatch object with the given secret string of lowercase letters. */
   public WordMatch(String word)
   {
      /* implementation not shown */
   }

   /** Returns a score for guess, as described in part (a).
    *  Precondition: 0 < guess.length() <= secret.length()
    */
   public int scoreGuess(String guess)
   {  /* to be implemented in part (a) */  }

   /** Returns the better of two guesses, as determined by scoreGuess and the rules for a
    *  tie-breaker that are described in part (b).
    *  Precondition: guess1 and guess2 contain all lowercase letters.
    *                guess1 is not the same as guess2.
    */
   public String findBetterGuess(String guess1, String guess2)
   {  /* to be implemented in part (b) */  }
}
```

(a) Write the `WordMatch` method `scoreGuess`. To determine the score to be returned, `scoreGuess` finds the number of times that `guess` occurs as a substring of `secret` and then multiplies that number by the square of the length of `guess`. Occurrences of `guess` may overlap within `secret`.

Assume that the length of `guess` is less than or equal to the length of `secret` and that `guess` is not an empty string.

The following examples show declarations of a `WordMatch` object. The tables show the outcomes of some possible calls to the `scoreGuess` method.

```
WordMatch game = new WordMatch("mississippi");
```

Value of `guess`	**Number of Substring Occurrences**	**Score Calculation**: **(Number of Substring Occurrences)** x **(Square of the Length of** `guess`)	**Return Value of** `game.scoreGuess(guess)`
`"i"`	4	`4 * 1 * 1 = 4`	`4`
`"iss"`	2	`2 * 3 * 3 = 18`	`18`
`"issipp"`	1	`1 * 6 * 6 = 36`	`36`
`"mississippi"`	1	`1 * 11 * 11 = 121`	`121`

```
WordMatch game = new WordMatch("aaaabb");
```

Value of `guess`	**Number of Substring Occurrences**	**Score Calculation**: **(Number of Substring Occurrences)** x **(Square of the Length of** `guess`)	**Return Value of** `game.scoreGuess(guess)`
`"a"`	4	`4 * 1 * 1 = 4`	`4`
`"aa"`	3	`3 * 2 * 2 = 12`	`12`
`"aaa"`	2	`2 * 3 * 3 = 18`	`18`
`"aabb"`	1	`1 * 4 * 4 = 16`	`16`
`"c"`	0	`0 * 1 * 1 = 0`	`0`

Complete the `scoreGuess` method.

```
/** Returns a score for guess, as described in part (a).
 *  Precondition: 0 < guess.length() <= secret.length()
 */
public int scoreGuess(String guess)
```

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 WordMatch

private String secret

public WordMatch(String word)
public int scoreGuess(String guess)
public String findBetterGuess(String guess1, String guess2)
```

(b) Write the `WordMatch` method `findBetterGuess`, which returns the better guess of its two `String` parameters, `guess1` and `guess2`. If the `scoreGuess` method returns different values for `guess1` and `guess2`, then the guess with the higher score is returned. If the `scoreGuess` method returns the same value for `guess1` and `guess2`, then the alphabetically greater guess is returned.

The following example shows a declaration of a `WordMatch` object and the outcomes of some possible calls to the `scoreGuess` and `findBetterGuess` methods.

```
WordMatch game = new WordMatch("concatenation");
```

Method Call	**Return Value**	**Explanation**
`game.scoreGuess("ten");`	`9`	`1 * 3 * 3`
`game.scoreGuess("nation");`	`36`	`1 * 6 * 6`
`game.findBetterGuess("ten", "nation");`	`"nation"`	Since `scoreGuess` returns `36` for `"nation"` and `9` for `"ten"`, the guess with the greater score, `"nation"`, is returned.
`game.scoreGuess("con");`	`9`	`1 * 3 * 3`
`game.scoreGuess("cat");`	`9`	`1 * 3 * 3`
`game.findBetterGuess("con", "cat");`	`"con"`	Since `scoreGuess` returns `9` for both `"con"` and `"cat"`, the alphabetically greater guess, `"con"`, is returned.

Complete method findBetterGuess.

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

```
/** Returns the better of two guesses, as determined by scoreGuess and the rules for a
 *  tie-breaker that are described in part (b).
 *  Precondition: guess1 and guess2 contain all lowercase letters.
 *                guess1 is not the same as guess2.
 */
public String findBetterGuess(String guess1, String guess2)
```

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 WordMatch

private String secret

public WordMatch(String word)
public int scoreGuess(String guess)
public String findBetterGuess(String guess1, String guess2)
```

2. 2011.4

2. In this question you will write two methods for a class `RouteCipher` that encrypts (puts into a coded form) a message by changing the order of the characters in the message. The route cipher fills a two-dimensional array with single-character substrings of the original message in row-major order, encrypting the message by retrieving the single-character substrings in column-major order.

For example, the word `"Surprise"` can be encrypted using a 2-row, 4-column array as follows.

Original Message: `"Surprise"` → Contents of Array → Encrypted Message: `"Sruirspe"`

"S"	"u"	"r"	"p"
"r"	"i"	"s"	"e"

An incomplete implementation of the `RouteCipher` class is shown below.

```
public class RouteCipher
{
  /** A two-dimensional array of single-character strings, instantiated in the constructor */
  private String[][] letterBlock;

  /** The number of rows of letterBlock, set by the constructor */
  private int numRows;

  /** The number of columns of letterBlock, set by the constructor */
  private int numCols;

  /** Places a string into letterBlock in row-major order.
   *  @param str the string to be processed
   *  Postcondition:
   *    if str.length() < numRows * numCols, "A" is placed in each unfilled cell
   *    if str.length() > numRows * numCols, trailing characters are ignored
   */
  private void fillBlock(String str)
  {  /* to be implemented in part (a) */  }

  /** Extracts encrypted string from letterBlock in column-major order.
   *  Precondition: letterBlock has been filled
   *  @return the encrypted string from letterBlock
   */
  private String encryptBlock()
  {  /* implementation not shown */  }

  /** Encrypts a message.
   *  @param message the string to be encrypted
   *  @return the encrypted message;
   *          if message is the empty string, returns the empty string
   */
  public String encryptMessage(String message)
  {  /* to be implemented in part (b) */  }

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

(a) Write the method `fillBlock` that fills the two-dimensional array `letterBlock` with one-character strings from the string passed as parameter `str`.

The array must be filled in row-major order—the first row is filled from left to right, then the second row is filled from left to right, and so on, until all rows are filled.

If the length of the parameter `str` is smaller than the number of elements of the array, the string `"A"` is placed in each of the unfilled cells. If the length of `str` is larger than the number of elements in the array, the trailing characters are ignored.

For example, if `letterBlock` has 3 rows and 5 columns and `str` is the string `"Meet at noon"`, the resulting contents of `letterBlock` would be as shown in the following table.

"M"	"e"	"e"	"t"	" "
"a"	"t"	" "	"n"	"o"
"o"	"n"	"A"	"A"	"A"

If `letterBlock` has 3 rows and 5 columns and `str` is the string `"Meet at midnight"`, the resulting contents of `letterBlock` would be as shown in the following table.

"M"	"e"	"e"	"t"	" "
"a"	"t"	" "	"m"	"i"
"d"	"n"	"i"	"g"	"h"

The following expression may be used to obtain a single-character string at position `k` of the string `str`.

```
str.substring(k, k + 1)
```

Complete method `fillBlock` below.

```
/** Places a string into letterBlock in row-major order.
 *  @param str  the string to be processed
 *  Postcondition:
 *     if str.length() < numRows * numCols, "A" is placed in each unfilled cell
 *     if str.length() > numRows * numCols, trailing characters are ignored
 */
private void fillBlock(String str)
```

(b) Write the method `encryptMessage` that encrypts its string parameter `message`. The method builds an encrypted version of `message` by repeatedly calling `fillBlock` with consecutive, nonoverlapping substrings of `message` and concatenating the results returned by a call to `encryptBlock` after each call to `fillBlock`. When all of `message` has been processed, the concatenated string is returned. Note that if `message` is the empty string, `encryptMessage` returns an empty string.

The following example shows the process carried out if `letterBlock` has 2 rows and 3 columns and `encryptMessage("Meet at midnight")` is executed.

Substring	`letterBlock` after Call to `fillBlock`	Value Returned by `encryptBlock`	Concatenated String

`"Meet a"`	`"M"` `"e"` `"e"`
	`"t"` `" "` `"a"`
		`"Mte ea"`	`"Mte ea"`

`"t midn"`	`"t"` `" "` `"m"`
	`"i"` `"d"` `"n"`
		`"ti dmn"`	`"Mte eati dmn"`

`"ight"`	`"i"` `"g"` `"h"`
	`"t"` `"A"` `"A"`
		`"itgAhA"`	`"Mte eati dmnitgAhA"`

In this example, the method returns the string `"Mte eati dmnitgAhA"`.

Assume that `fillBlock` and `encryptBlock` methods work as specified. Solutions that reimplement the functionality of one or both of these methods will not receive full credit.

Complete method `encryptMessage` below.

```
/** Encrypts a message.
 *  @param message  the string to be encrypted
 *  @return  the encrypted message;
 *           if message is the empty string, returns the empty string
 */
public String encryptMessage(String message)
```

3. 2017.3

26. This question involves analyzing and modifying a string. The following `Phrase` class maintains a phrase in an instance variable and has methods that access and make changes to the phrase. You will write two methods of the `Phrase` class.

```
public class Phrase
{
   private String currentPhrase;

   /** Constructs a new Phrase object. */
   public Phrase(String p)
   {  currentPhrase = p;  }


   /** Returns the index of the nth occurrence of str in the current phrase;
    *  returns -1 if the nth occurrence does not exist.
    *  Precondition: str.length() > 0 and n > 0
    *  Postcondition: the current phrase is not modified.
    */
   public int findNthOccurrence(String str, int n)
   {   /* implementation not shown */   }


   /** Modifies the current phrase by replacing the nth occurrence of str with repl.
    *  If the nth occurrence does not exist, the current phrase is unchanged.
    *  Precondition: str.length() > 0 and n > 0
    */
   public void replaceNthOccurrence(String str, int n, String repl)
   {   /* to be implemented in part (a) */   }


   /** Returns the index of the last occurrence of str in the current phrase;
    *  returns -1 if str is not found.
    *  Precondition: str.length() > 0
    *  Postcondition: the current phrase is not modified.
    */
   public int findLastOccurrence(String str)
   {   /* to be implemented in part (b) */   }


   /** Returns a string containing the current phrase. */
   public String toString()
   {  return currentPhrase;  }
}
```

(a) Write the `Phrase` method `replaceNthOccurrence`, which will replace the `nth` occurrence of the string `str` with the string `repl`. If the `nth` occurrence does not exist, `currentPhrase` remains unchanged.

Several examples of the behavior of the method `replaceNthOccurrence` are shown below.

Code segments	Output produced

`Phrase phrase1 = new Phrase("A cat ate late.");`
`phrase1.replaceNthOccurrence("at", 1, "rane");`
`System.out.println(phrase1);`
	`A crane ate late.`

`Phrase phrase2 = new Phrase("A cat ate late.");`
`phrase2.replaceNthOccurrence("at", 6, "xx");`
`System.out.println(phrase2);`
	`A cat ate late.`

`Phrase phrase3 = new Phrase("A cat ate late.");`
`phrase3.replaceNthOccurrence("bat", 2, "xx");`
`System.out.println(phrase3);`
	`A cat ate late.`

`Phrase phrase4 = new Phrase("aaaa");`
`phrase4.replaceNthOccurrence("aa", 1, "xx");`
`System.out.println(phrase4);`
	`xxaa`

`Phrase phrase5 = new Phrase("aaaa");`
`phrase5.replaceNthOccurrence("aa", 2, "bbb");`
`System.out.println(phrase5);`
	`abbba`

Class information for this question

```
public class Phrase

private String currentPhrase
public Phrase(String p)
public int findNthOccurrence(String str, int n)
public void replaceNthOccurrence(String str, int n, String repl)
public int findLastOccurrence(String str)
public String toString()
```

The `Phrase` class includes the method `findNthOccurrence`, which returns the `nth` occurrence of a given string. You must use `findNthOccurrence` appropriately to receive full credit.

Complete method `replaceNthOccurrence` below.

```
/** Modifies the current phrase by replacing the nth occurrence of str with repl.
 *  If the nth occurrence does not exist, the current phrase is unchanged.
 *  Precondition: str.length() > 0 and n > 0
 */
public void replaceNthOccurrence(String str, int n, String repl)
```

(b) Write the `Phrase` method `findLastOccurrence`. This method finds and returns the index of the last occurrence of a given string in `currentPhrase`. If the given string is not found, `-1` is returned. The following tables show several examples of the behavior of the method `findLastOccurrence`.

```
Phrase phrase1 = new Phrase("A cat ate late.");
```

Method call	Value returned
`phrase1.findLastOccurrence("at")`	11
`phrase1.findLastOccurrence("cat")`	2
`phrase1.findLastOccurrence("bat")`	-1

Class information for this question

```
public class Phrase

private String currentPhrase
public Phrase(String p)
public int findNthOccurrence(String str, int n)
public void replaceNthOccurrence(String str, int n, String repl)
public int findLastOccurrence(String str)
public String toString()
```

You must use `findNthOccurrence` appropriately to receive full credit.

Complete method `findLastOccurrence` below.

```
/** Returns the index of the last occurrence of str in the current phrase;
 *  returns -1 if str is not found.
 *  Precondition: str.length() > 0
 *  Postcondition: the current phrase is not modified.
 */
public int findLastOccurrence(String str)
```

Class

1. 2019(2).2

27. 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.

2. 2019(1). 2

2. This question involves the creation and use of a spinner to generate random numbers in a game. A `GameSpinner` object represents a spinner with a given number of sectors, all equal in size. The `GameSpinner` class supports the following behaviors.

- Creating a new spinner with a specified number of sectors
- Spinning a spinner and reporting the result
- Reporting the length of the *current run*, the number of consecutive spins that are the same as the most recent spin

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

Statements	**Value Returned (blank if no value returned)**	**Comment**
`GameSpinner g = new GameSpinner(4);`		Creates a new spinner with four sectors
`g.currentRun();`	`0`	Returns the length of the current run. The length of the current run is initially `0` because no spins have occurred.
`g.spin();`	`3`	Returns a random integer between `1` and `4`, inclusive. In this case, `3` is returned.
`g.currentRun();`	`1`	The length of the current run is `1` because there has been one spin of `3` so far.
`g.spin();`	`3`	Returns a random integer between `1` and `4`, inclusive. In this case, `3` is returned.
`g.currentRun();`	`2`	The length of the current run is `2` because there have been two `3`s in a row.
`g.spin();`	`4`	Returns a random integer between `1` and `4`, inclusive. In this case, `4` is returned.
`g.currentRun();`	`1`	The length of the current run is `1` because the spin of `4` is different from the value of the spin in the previous run of two `3`s.
`g.spin();`	`3`	Returns a random integer between `1` and `4`, inclusive. In this case, `3` is returned.
`g.currentRun();`	`1`	The length of the current run is `1` because the spin of `3` is different from the value of the spin in the previous run of one `4`.
`g.spin();`	`1`	Returns a random integer between `1` and `4`, inclusive. In this case, `1` is returned.
`g.spin();`	`1`	Returns a random integer between `1` and `4`, inclusive. In this case, `1` is returned.
`g.spin();`	`1`	Returns a random integer between `1` and `4`, inclusive. In this case, `1` is returned.
`g.currentRun();`	`3`	The length of the current run is `3` because there have been three consecutive `1`s since the previous run of one `3`.

3. 2021.2

3. 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.

4. 2015.2

4. 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 `getHint`	**String returned**
`puzzle.getHint("AAAAA")`	`"+A+++"`
`puzzle.getHint("HELLO")`	`"H****"`
`puzzle.getHint("HEART")`	`"H*++*"`
`puzzle.getHint("HARMS")`	`"HAR*S"`
`puzzle.getHint("HARPS")`	`"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.

5. 2016.1

5. 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 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.

(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)
```

6. 2010.2

6. An APLine is a line defined by the equation $ax + by + c = 0$, where a is not equal to zero, b is not equal to zero, and a, b, and c are all integers. The slope of an APLine is defined to be the double value $-a/b$. A point (represented by integers x and y) is on an APLine if the equation of the APLine is satisfied when those x and y values are substituted into the equation. That is, a point represented by x and y is on the line if $ax + by + c$ is equal to 0. Examples of two APLine equations are shown in the following table.

Equation	**Slope (–*a* / *b*)**	**Is point (5, -2) on the line?**
$5x + 4y - 17 = 0$	-5 / 4 = -1.25	Yes, because 5(5) + 4(-2) + (-17) = 0
$-25x + 40y + 30 = 0$	25 / 40 = 0.625	No, because -25(5) + 40(-2) + 30 ≠ 0

Assume that the following code segment appears in a class other than APLine. The code segment shows an example of using the APLine class to represent the two equations shown in the table.

```
APLine line1 = new APLine(5, 4, -17);
double slope1 = line1.getSlope();        // slope1 is assigned -1.25
boolean onLine1 = line1.isOnLine(5, -2); // true because 5(5) + 4(-2) + (-17) = 0


APLine line2 = new APLine(-25, 40, 30);
double slope2 = line2.getSlope();        // slope2 is assigned 0.625
boolean onLine2 = line2.isOnLine(5, -2); // false because -25(5) + 40(-2) + 30 ≠ 0
```

Write the APLine class. Your implementation must include a constructor that has three integer parameters that represent a, b, and c, in that order. You may assume that the values of the parameters representing a and b are not zero. It must also include a method getSlope that calculates and returns the slope of the line, and a method isOnLine that returns true if the point represented by its two parameters (x and y, in that order) is on the APLine and returns false otherwise. Your class must produce the indicated results when invoked by the code segment given above. You may ignore any issues related to integer overflow.

7. 2022.2

7. 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.

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.

ArrayList FRQ

1. 2019(2).3

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

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

   `(x + y) * 5` is a valid mathematical expression.

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

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

   `<B> Make this text bold </B>` is valid HTML.

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

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

```
public class Delimiters
{

   /** The open and close delimiters. */
   private String openDel;
   private String closeDel;

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

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

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

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

}
```

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

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

Example 1

openDel:	"("			
closeDel:	")"			
tokens:	"("	"x + y"	")"	" * 5"
ArrayList of delimiters:	"("	")"		

Example 2

openDel:	"<q>"				
closeDel:	"</q>"				
tokens:	"<q>"	"yy"	"</q>"	"zz"	"</q>"
ArrayList of delimiters:	"<q>"	"</q>"	"</q>"		

Class information for this question

```
public class Delimiters

private String openDel
private String closeDel

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

Complete method getDelimitersList below.

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

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

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

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

Example 1

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

"<sup>"	"<sup>"	"</sup>"	"<sup>"	"</sup>"	"</sup>"

Example 2

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

"<sup>"	"</sup>"	"</sup>"	"<sup>"

↑

When starting from the left, at this point, condition 1 is violated.

Example 3

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

"</sup>"

↑

At this point, condition 1 is violated.

Example 4

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

"<sup>"	"<sup>"	"</sup>"

Class information for this question

```
public class Delimiters

private String openDel
private String closeDel

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

Complete method `isBalanced` below.

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

2. 2010.1

2. An organization raises money by selling boxes of cookies. A cookie order specifies the variety of cookie and the number of boxes ordered. The declaration of the `CookieOrder` class is shown below.

```
public class CookieOrder
{
   /** Constructs a new CookieOrder object. */
   public CookieOrder(String variety, int numBoxes)
   {  /* implementation not shown */  }

   /** @return the variety of cookie being ordered
    */
   public String getVariety()
   {  /* implementation not shown */  }

   /** @return the number of boxes being ordered
    */
   public int getNumBoxes()
   {  /* implementation not shown */  }

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

The MasterOrder class maintains a list of the cookies to be purchased. The declaration of the MasterOrder class is shown below.

```
public class MasterOrder
{
  /** The list of all cookie orders */
  private List<CookieOrder> orders;

  /** Constructs a new MasterOrder object. */
  public MasterOrder()
  {  orders = new ArrayList<CookieOrder>();  }

  /** Adds theOrder to the master order.
   *  @param theOrder the cookie order to add to the master order
   */
  public void addOrder(CookieOrder theOrder)
  {  orders.add(theOrder);  }

  /** @return the sum of the number of boxes of all of the cookie orders
   */
  public int getTotalBoxes()
  {   /* to be implemented in part (a) */   }

  /** Removes all cookie orders from the master order that have the same variety of
   *  cookie as cookieVar and returns the total number of boxes that were removed.
   *  @param cookieVar the variety of cookies to remove from the master order
   *  @return the total number of boxes of cookieVar in the cookie orders removed
   */
  public int removeVariety(String cookieVar)
  {   /* to be implemented in part (b) */   }

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

(a) The getTotalBoxes method computes and returns the sum of the number of boxes of all cookie orders. If there are no cookie orders in the master order, the method returns 0.

Complete method getTotalBoxes below.

```
  /** @return the sum of the number of boxes of all of the cookie orders
   */
  public int getTotalBoxes()
```

(b) The removeVariety method updates the master order by removing all of the cookie orders in which the variety of cookie matches the parameter cookieVar. The master order may contain zero or more cookie orders with the same variety as cookieVar. The method returns the total number of boxes removed from the master order.

For example, consider the following code segment.

```
MasterOrder goodies = new MasterOrder();
goodies.addOrder(new CookieOrder("Chocolate Chip", 1));
goodies.addOrder(new CookieOrder("Shortbread", 5));
goodies.addOrder(new CookieOrder("Macaroon", 2));
goodies.addOrder(new CookieOrder("Chocolate Chip", 3));
```

After the code segment has executed, the contents of the master order are as shown in the following table.

"Chocolate Chip" 1	"Shortbread" 5	"Macaroon" 2	"Chocolate Chip" 3

The method call goodies.removeVariety("Chocolate Chip") returns 4 because there were two Chocolate Chip cookie orders totaling 4 boxes. The master order is modified as shown below.

"Shortbread" 5	"Macaroon" 2

The method call goodies.removeVariety("Brownie") returns 0 and does <u>not</u> change the master order.

Complete method removeVariety below.

```
/** Removes all cookie orders from the master order that have the same variety of
 *  cookie as cookieVar and returns the total number of boxes that were removed.
 *  @param cookieVar the variety of cookies to remove from the master order
 *  @return the total number of boxes of cookieVar in the cookie orders removed
 */
public int removeVariety(String cookieVar)
```

4. 2021.3

4. A high school club maintains information about its members in a `MemberInfo` object. A `MemberInfo` object stores a club member's name, year of graduation, and whether or not the club member is in *good standing*. A member who is in good standing has fulfilled all the responsibilities of club membership.

A partial declaration of the `MemberInfo` class is shown below.

```
public class MemberInfo
{
   /** Constructs a MemberInfo object for the club member with name name,
    *  graduation year gradYear, and standing hasGoodStanding.
    */
   public MemberInfo(String name, int gradYear, boolean hasGoodStanding)
   {  /* implementation not shown */  }

   /** Returns the graduation year of the club member. */
   public int getGradYear()
   {  /* implementation not shown */  }

   /** Returns true if the member is in good standing and false otherwise. */
   public boolean inGoodStanding()
   {  /* implementation not shown */  }

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

The `ClubMembers` class maintains a list of current club members. The declaration of the `ClubMembers` class is shown below.

```
public class ClubMembers
{
   private ArrayList<MemberInfo> memberList;

   /** Adds new club members to memberList, as described in part (a).
    *  Precondition: names is a non-empty array.
    */
   public void addMembers(String[] names, int gradYear)
   {  /* to be implemented in part (a) */  }

   /** Removes members who have graduated and returns a list of members who have graduated
    *  and are in good standing, as described in part (b).
    */

   public ArrayList<MemberInfo> removeMembers(int year)
   {  /* to be implemented in part (b) */  }// There may be instance
   variables, constructors, and methods that are not shown.

}
```

(a) Write the `ClubMembers` method `addMembers`, which takes two parameters. The first parameter is a `String` array containing the names of new club members to be added. The second parameter is the graduation year of all the new club members. The method adds the new members to the `memberList` instance variable. The names can be added in any order. All members added are initially in good standing and share the same graduation year, `gradYear`.

Complete the `addMembers` method.

```
/** Adds new club members to memberList, as described in part (a).
 *  Precondition: names is a non-empty array.
 */
public void addMembers(String[] names, int gradYear)
```

(b) Write the `ClubMembers` method `removeMembers`, which takes the following actions.

- Returns a list of all students who have graduated and are in good standing. A member has graduated if the member's graduation year is less than or equal to the method's `year` parameter. If no members meet these criteria, an empty list is returned.
- Removes from `memberList` all members who have graduated, regardless of whether or not they are in good standing.

The following example illustrates the results of a call to `removeMembers`.

The `ArrayList memberList` before the method call `removeMembers(2018)`:

"SMITH, JANE"	"FOX, STEVE"	"XIN, MICHAEL"	"GARCIA, MARIA"
2019	2018	2017	2020
false	true	false	true

The `ArrayList memberList` after the method call `removeMembers(2018)`:

"SMITH, JANE"	"GARCIA, MARIA"
2019	2020
false	true

The `ArrayList` returned by the method call `removeMembers(2018)`:

"FOX, STEVE"
2018
true

Complete the `removeMembers` method.

```
/** Removes members who have graduated and returns a list of members who have graduated and are
 *  in good standing, as described in part (b).
 */
public ArrayList<MemberInfo> removeMembers(int year)
```

Class information for this question

```
public class MemberInfo

public MemberInfo(String name, int gradYear, boolean hasGoodStanding)
public int getGradYear()
public boolean inGoodStanding()

public class ClubMembers

private ArrayList<MemberInfo> memberList

public void addMembers(String[] names, int gradYear)
public ArrayList<MemberInfo> removeMembers(int year)
```

4. 2022.3

4. 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.
}
```

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) */  }
}
```

(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.

0	1	2	3	4
4 "Good! Thx"	3 "OK site"	5 "Great!"	2 "Poor! Bad."	3 ""

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()
```

(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).

0	1	2	3	4
4 "Good! Thx"	3 "OK site"	5 "Great!"	2 "Poor! Bad."	3 ""

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.

"0-Good! Thx."	"2-Great!"	"3-Poor! Bad."

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.
 *  Postcondition: allReviews is unchanged.
 */
public ArrayList<String> collectComments()
```

5. 2018.2

5. This question involves reasoning about pairs of words that are represented by the following `WordPair` class.

```
public class WordPair
{

    /** Constructs a WordPair object. */
    public WordPair(String first, String second)
    {  /* implementation not shown */  }


    /** Returns the first string of this WordPair object. */
    public String getFirst()
    {  /* implementation not shown */  }


    /** Returns the second string of this WordPair object. */
    public String getSecond()
    {  /* implementation not shown */  }

}
```

You will implement the constructor and another method for the following `WordPairList` class.

```
public class WordPairList
{

    /** The list of word pairs, initialized by the constructor. */
    private ArrayList<WordPair> allPairs;


    /** Constructs a WordPairList object as described in part (a).
     *  Precondition: words.length >= 2
     */
    public WordPairList(String[] words)
    { /* to be implemented in part (a) */ }


    /** Returns the number of matches as described in part (b).
     */
    public int numMatches()
    { /* to be implemented in part (b) */ }

}
```

(a) Write the constructor for the `WordPairList` class. The constructor takes an array of strings `words` as a parameter and initializes the instance variable `allPairs` to an `ArrayList` of `WordPair` objects.

A `WordPair` object consists of a word from the array paired with a word that appears later in the array. The `allPairs` list contains `WordPair` objects `(words[i], words[j])` for every `i` and `j`, where 0 ≤ `i` < `j` < `words.length`. Each `WordPair` object is added exactly once to the list.

The following examples illustrate two different `WordPairList` objects.

Example 1

```
String[] wordNums = {"one", "two", "three"};
WordPairList exampleOne = new WordPairList(wordNums);
```

After the code segment has executed, the `allPairs` instance variable of `exampleOne` will contain the following `WordPair` objects in some order.

```
  ("one", "two"), ("one", "three"), ("two", "three")
```

Example 2

```
String[] phrase = {"the", "more", "the", "merrier"};
WordPairList exampleTwo = new WordPairList(phrase);
```

After the code segment has executed, the `allPairs` instance variable of `exampleTwo` will contain the following `WordPair` objects in some order.

```
  ("the", "more"), ("the", "the"), ("the", "merrier"),
  ("more", "the"), ("more", "merrier"), ("the", "merrier")
```

Class information for this question

```
public class WordPair

public WordPair(String first, String second)
public String getFirst()
public String getSecond()

public class WordPairList

private ArrayList<WordPair> allPairs

public WordPairList(String[] words)
public int numMatches()
```

Complete the `WordPairList` constructor below.

```
/** Constructs a WordPairList object as described in part (a).
 *  Precondition: words.length >= 2
 */
public WordPairList(String[] words)
```

(b) Write the `WordPairList` method `numMatches`. This method returns the number of `WordPair` objects in `allPairs` for which the two strings match.

For example, the following code segment creates a `WordPairList` object.

```
String[] moreWords = {"the", "red", "fox", "the", "red"};
WordPairList exampleThree = new WordPairList(moreWords);
```

After the code segment has executed, the `allPairs` instance variable of `exampleThree` will contain the following `WordPair` objects in some order. The pairs in which the first string matches the second string are shaded for illustration.

```
  ("the", "red"), ("the", "fox"), ("the", "the"),
  ("the", "red"), ("red", "fox"), ("red", "the"),
  ("red", "red"), ("fox", "the"), ("fox", "red"),
  ("the", "red")
```

The call `exampleThree.numMatches()` should return `2`.

Class information for this question

```
public class WordPair

public WordPair(String first, String second)
public String getFirst()
public String getSecond()

public class WordPairList

private ArrayList<WordPair> allPairs

public WordPairList(String[] words)
public int numMatches()
```

Complete method `numMatches` below.

```
/** Returns the number of matches as described in part (b).
 */
public int numMatches()
```

6. 2012.1

6. A mountain climbing club maintains a record of the climbs that its members have made. Information about a climb includes the name of the mountain peak and the amount of time it took to reach the top. The information is contained in the `ClimbInfo` class as declared below.

```
public class ClimbInfo
{
  /** Creates a ClimbInfo object with name peakName and time climbTime.
   *  @param peakName  the name of the mountain peak
   *  @param climbTime  the number of minutes taken to complete the climb
   */
  public ClimbInfo(String peakName, int climbTime)
  {  /* implementation not shown */  }


  /** @return  the name of the mountain peak
   */
  public String getName()
  {  /* implementation not shown */  }


  /** @return  the number of minutes taken to complete the climb
   */
  public int getTime()
  {  /* implementation not shown */  }

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

The ClimbingClub class maintains a list of the climbs made by members of the club. The declaration of the ClimbingClub class is shown below. You will write two different implementations of the addClimb method. You will also answer two questions about an implementation of the distinctPeakNames method.

```
public class ClimbingClub
{
   /**  The list of climbs completed by members of the club.
    *   Guaranteed not to be null.  Contains only non-null references.
    */
   private List<ClimbInfo> climbList;


   /**  Creates a new ClimbingClub object. */
   public ClimbingClub()
   {   climbList = new ArrayList<ClimbInfo>();   }


   /**  Adds a new climb with name peakName and time climbTime to the list of climbs.
    *   @param peakName the name of the mountain peak climbed
    *   @param climbTime the number of minutes taken to complete the climb
    */
   public void addClimb(String peakName, int climbTime)
   {   /* to be implemented in part (a) with ClimbInfo objects in the order they were added    */
       /* to be implemented in part (b) with ClimbInfo objects in alphabetical order by name */
   }


   /** @return the number of distinct names in the list of climbs */
   public int distinctPeakNames()
   {   /* implementation shown in part (c) */   }

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

(a) Write an implementation of the `ClimbingClub` method `addClimb` that stores the `ClimbInfo` objects in the order they were added. This implementation of `addClimb` should create a new `ClimbInfo` object with the given name and time. It appends a reference to that object to the end of `climbList`. For example, consider the following code segment.

```
ClimbingClub hikerClub = new ClimbingClub();
hikerClub.addClimb("Monadnock", 274);
hikerClub.addClimb("Whiteface", 301);
hikerClub.addClimb("Algonquin", 225);
hikerClub.addClimb("Monadnock", 344);
```

When the code segment has completed executing, the instance variable `climbList` would contain the following entries.

Peak Name	"Monadnock"	"Whiteface"	"Algonquin"	"Monadnock"
Climb Time	274	301	225	344

Information repeated from the beginning of the question

```
public class ClimbInfo

public ClimbInfo(String peakName, int climbTime)
public String getName()
public int getTime()


public class ClimbingClub

private List<ClimbInfo> climbList
public void addClimb(String peakName, int climbTime)
public int distinctPeakNames()
```

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Complete method `addClimb` below.

```
/** Adds a new climb with name peakName and time climbTime to the list of climbs.
 *  @param peakName the name of the mountain peak climbed
 *  @param climbTime the number of minutes taken to complete the climb
 *  Postcondition: The new entry is at the end of climbList;
 *                 The order of the remaining entries is unchanged.
 */
public void addClimb(String peakName, int climbTime)
```

(b) Write an implementation of the `ClimbingClub` method `addClimb` that stores the elements of `climbList` in alphabetical order by name (as determined by the `compareTo` method of the `String` class). This implementation of `addClimb` should create a new `ClimbInfo` object with the given name and time and then insert the object into the appropriate position in `climbList`. Entries that have the same name will be grouped together and can appear in any order within the group. For example, consider the following code segment.

```
ClimbingClub hikerClub = new ClimbingClub();
hikerClub.addClimb("Monadnock", 274);
hikerClub.addClimb("Whiteface", 301);
hikerClub.addClimb("Algonquin", 225);
hikerClub.addClimb("Monadnock", 344);
```

When the code segment has completed execution, the instance variable `climbList` would contain the following entries in either of the orders shown below.

Peak Name	"Algonquin"	"Monadnock"	"Monadnock"	"Whiteface"
Climb Time	225	344	274	301

OR

Peak Name	"Algonquin"	"Monadnock"	"Monadnock"	"Whiteface"
Climb Time	225	274	344	301

You may assume that `climbList` is in alphabetical order by name when the method is called. When the method has completed execution, `climbList` should still be in alphabetical order by name.

Information repeated from the beginning of the question

```
public class ClimbInfo

public ClimbInfo(String peakName, int climbTime)
public String getName()
public int getTime()


public class ClimbingClub

private List<ClimbInfo> climbList
public void addClimb(String peakName, int climbTime)
public int distinctPeakNames()
```

Complete method addClimb below.

```
/** Adds a new climb with name peakName and time climbTime to the list of climbs.
 *  Alphabetical order is determined by the compareTo method of the String class.
 *  @param peakName the name of the mountain peak climbed
 *  @param climbTime the number of minutes taken to complete the climb
 *  Precondition:  entries in climbList are in alphabetical order by name.
 *  Postcondition: entries in climbList are in alphabetical order by name.
 */
public void addClimb(String peakName, int climbTime)
```

(c) The ClimbingClub method distinctPeakNames is intended to return the number of different names in climbList. For example, after the following code segment has completed execution, the value of the variable numNames would be 3.

```
ClimbingClub hikerClub = new ClimbingClub();
hikerClub.addClimb("Monadnock", 274);
hikerClub.addClimb("Whiteface", 301);
hikerClub.addClimb("Algonquin", 225);
hikerClub.addClimb("Monadnock", 344);
int numNames = hikerClub.distinctPeakNames();
```

Consider the following implementation of method distinctPeakNames.

```
/** @return the number of distinct names in the list of climbs */
public int distinctPeakNames()
{
  if (climbList.size() == 0)
  {
    return 0;
  }

  ClimbInfo currInfo = climbList.get(0);
  String prevName = currInfo.getName();
  String currName = null;
  int numNames = 1;

  for (int k = 1; k < climbList.size(); k++)
  {
    currInfo = climbList.get(k);
    currName = currInfo.getName();
    if (prevName.compareTo(currName) != 0)
    {
       numNames++;
       prevName = currName;
    }
  }
  return numNames;
}
```

Assume that `addClimb` works as specified, regardless of what you wrote in parts (a) and (b).

(i) Does this implementation of the `distinctPeakNames` method work as intended when the `addClimb` method stores the `ClimbInfo` objects in the order they were added as described in part (a)?

Circle one of the answers below.

YES NO

(ii) Does this implementation of the `distinctPeakNames` method work as intended when the `addClimb` method stores the `ClimbInfo` objects in alphabetical order by name as described in part (b)?

Circle one of the answers below.

YES NO

7. 2013.1

7. A music Web site keeps track of downloaded music. For each download, the site uses a `DownloadInfo` object to store a song's title and the number of times it has been downloaded. A partial declaration for the `DownloadInfo` class is shown below.

```
public class DownloadInfo
{
   /**  Creates a new instance with the given unique title and sets the
    *   number of times downloaded to 1.
    *   @param title  the unique title of the downloaded song
    */
   public DownloadInfo(String title)
   {   /*  implementation not shown  */   }


   /** @return  the title  */
   public String getTitle()
   {   /*  implementation not shown  */   }


   /**  Increment the number times downloaded by 1  */
   public void incrementTimesDownloaded()
   {   /*  implementation not shown  */   }

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

The list of downloaded information is stored in a MusicDownloads object. A partial declaration for the MusicDownloads class is shown below.

```
public class MusicDownloads
{
  /** The list of downloaded information.
   *  Guaranteed not to be null and not to contain duplicate titles.
   */
  private List<DownloadInfo> downloadList;


  /** Creates the list of downloaded information. */
  public MusicDownloads()
  {  downloadList = new ArrayList<DownloadInfo>();  }


  /** Returns a reference to the DownloadInfo object with the requested title if it exists.
   *  @param title the requested title
   *  @return a reference to the DownloadInfo object with the
   *          title that matches the parameter title if it exists in the list;
   *          null otherwise.
   *  Postcondition:
   *    - no changes were made to downloadList.
   */
  public DownloadInfo getDownloadInfo(String title)
  {  /* to be implemented in part (a) */  }


  /** Updates downloadList with information from titles.
   *  @param titles a list of song titles
   *  Postcondition:
   *    - there are no duplicate titles in downloadList.
   *    - no entries were removed from downloadList.
   *    - all songs in titles are represented in downloadList.
   *    - for each existing entry in downloadList, the download count is increased by
   *        the number of times its title appeared in titles.
   *    - the order of the existing entries in downloadList is not changed.
   *    - the first time an object with a title from titles is added to downloadList, it
   *        is added to the end of the list.
   *    - new entries in downloadList appear in the same order
   *        in which they first appear in titles.
   *    - for each new entry in downloadList, the download count is equal to
   *        the number of times its title appeared in titles.
   */
  public void updateDownloads(List<String> titles)
  {  /* to be implemented in part (b) */  }

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

(a) Write the `MusicDownloads` method `getDownloadInfo`, which returns a reference to a `DownloadInfo` object if an object with a title that matches the parameter `title` exists in the `downloadList`. If no song in `downloadList` has a title that matches the parameter `title`, the method returns `null`.

For example, suppose variable `webMusicA` refers to an instance of `MusicDownloads` and that the table below represents the contents of `downloadList`. The list contains three `DownloadInfo` objects. The object at position 0 has a title of "Hey Jude" and a download count of 5. The object at position 1 has a title of "Soul Sister" and a download count of 3. The object at position 2 has a title of "Aqualung" and a download count of 10.

0	1	2
"Hey Jude" 5	"Soul Sister" 3	"Aqualung" 10

The call `webMusicA.getDownloadInfo("Aqualung")` returns a reference to the object in position 2 of the list.

The call `webMusicA.getDownloadInfo("Happy Birthday")` returns `null` because there are no `DownloadInfo` objects with that title in the list.

Class information repeated from the beginning of the question

```
public class DownloadInfo

public DownloadInfo(String title)
public String getTitle()
public void incrementTimesDownloaded()

public class MusicDownloads

private List<DownloadInfo> downloadList
public DownloadInfo getDownloadInfo(String title)
public void updateDownloads(List<String> titles)
```

Complete method `getDownloadInfo` below.

```
/** Returns a reference to the DownloadInfo object with the requested title if it exists.
 *  @param title the requested title
 *  @return a reference to the DownloadInfo object with the
 *          title that matches the parameter title if it exists in the list;
 *          null otherwise.
 *  Postcondition:
 *    - no changes were made to downloadList.
 */
public DownloadInfo getDownloadInfo(String title)
```

(b) Write the `MusicDownloads` method `updateDownloads`, which takes a list of song titles as a parameter. For each title in the list, the method updates `downloadList`, either by incrementing the download count if a `DownloadInfo` object with the same title exists, or by adding a new `DownloadInfo` object with that title and a download count of 1 to the end of the list. When a new `DownloadInfo` object is added to the end of the list, the order of the already existing entries in `downloadList` remains unchanged.

For example, suppose variable `webMusicB` refers to an instance of `MusicDownloads` and that the table below represents the contents of the instance variable `downloadList`.

0	1	2
"Hey Jude" 5	"Soul Sister" 3	"Aqualung" 10

Assume that the variable `List<String> songTitles` has been defined and contains the following entries.

```
{"Lights", "Aqualung", "Soul Sister", "Go Now", "Lights", "Soul Sister"}
```

The call `webMusicB.updateDownloads(songTitles)` results in the following `downloadList` with incremented download counts for the objects with titles of "Soul Sister" and "Aqualung". It also has a new `DownloadInfo` object with a title of "Lights" and a download count of 2, and another `DownloadInfo` object with a title of "Go Now" and a download count of 1. The order of the already existing entries remains unchanged.

0	1	2	3	4
"Hey Jude" 5	"Soul Sister" 5	"Aqualung" 11	"Lights" 2	"Go Now" 1

Class information repeated from the beginning of the question

```
public class DownloadInfo

public DownloadInfo(String title)
public String getTitle()
public void incrementTimesDownloaded()

public class MusicDownloads

private List<DownloadInfo> downloadList
public DownloadInfo getDownloadInfo(String title)
public void updateDownloads(List<String> titles)
```

In writing your solution, you must use the `getDownloadInfo` method. Assume that `getDownloadInfo` works as specified, regardless of what you wrote for part (a).

Complete method `updateDownloads` below.

```
/** Updates downloadList with information from titles.
 *  @param titles a list of song titles
 *  Postcondition:
 *    – there are no duplicate titles in downloadList.
 *    – no entries were removed from downloadList.
 *    – all songs in titles are represented in downloadList.
 *    – for each existing entry in downloadList, the download count is increased by
 *         the number of times its title appeared in titles.
 *    – the order of the existing entries in downloadList is not changed.
 *    – the first time an object with a title from titles is added to downloadList, it
 *         is added to the end of the list.
 *    – new entries in downloadList appear in the same order
 *         in which they first appear in titles.
 *    – for each new entry in downloadList, the download count is equal to
 *         the number of times its title appeared in titles.
 */
public void updateDownloads(List<String> titles )
```

8. 2017.1

8. 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) */   }
}
```

(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)
```

(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`

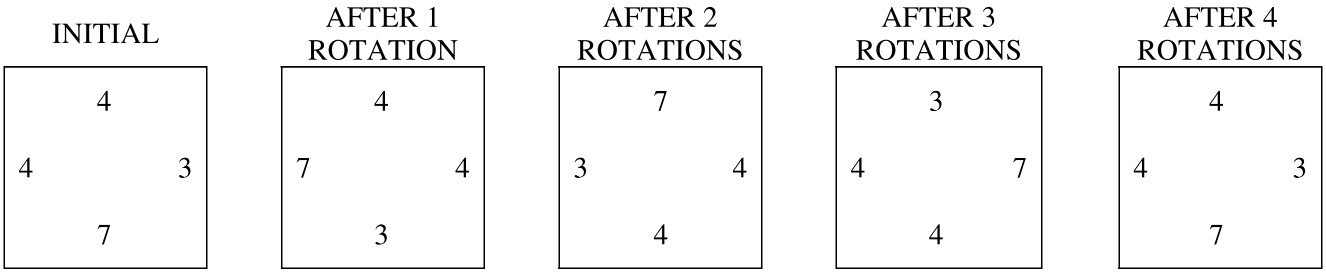
WRITE YOUR SOLUTION ON THE NEXT PAGE.

Complete method `isStrictlyIncreasing` below.

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

9. 2009.4

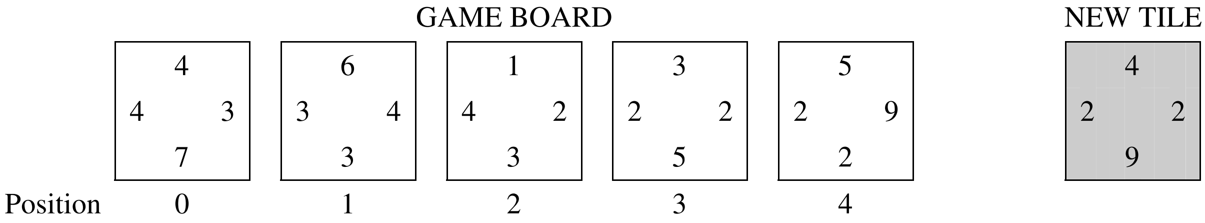
9. A game uses square tiles that have numbers on their sides. Each tile is labeled with a number on each of its four sides and may be rotated clockwise, as illustrated below.



The tiles are represented by the `NumberTile` class, as given below.

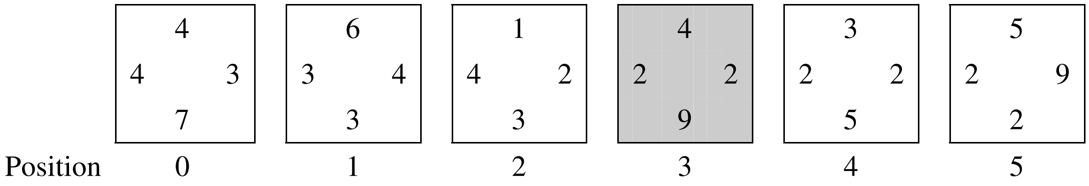
```
public class NumberTile
{
   /** Rotates the tile 90 degrees clockwise
     */
   public void rotate()
   {   /* implementation not shown */   }


   /** @return value at left edge of tile
     */
   public int getLeft()
   {   /* implementation not shown */   }


   /** @return value at right edge of tile
     */
   public int getRight()
   {   /* implementation not shown */   }

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

Tiles are placed on a game board so that the adjoining sides of adjacent tiles have the same number. The following figure illustrates an arrangement of tiles and shows a new tile that is to be placed on the game board.



In its original orientation, the new tile can be inserted between the tiles at positions 2 and 3 or between the tiles at positions 3 and 4. If the new tile is rotated once, it can be inserted before the tile at position 0 (the first tile) or after the tile at position 4 (the last tile). Assume that the new tile, in its original orientation, is inserted between the tiles at positions 2 and 3. As a result of the insertion, the tiles at positions 3 and 4 are moved one location to the right, and the new tile is inserted at position 3, as shown below.

GAME BOARD AFTER INSERTING TILE

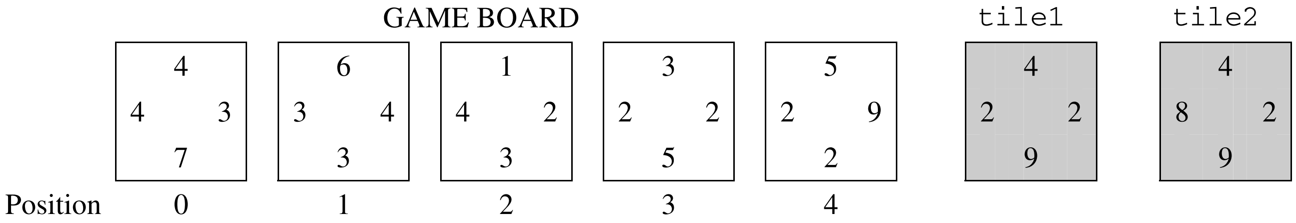


A partial definition of the `TileGame` class is given below.

```
public class TileGame
{
  /** represents the game board; guaranteed never to be null */
  private ArrayList<NumberTile> board;

  public TileGame()
  { board = new ArrayList<NumberTile>(); }

  /** Determines where to insert tile, in its current orientation, into game board
   *  @param tile the tile to be placed on the game board
   *  @return the position of tile where tile is to be inserted:
   *             0 if the board is empty;
   *            -1 if tile does not fit in front, at end, or between any existing tiles;
   *            otherwise, 0 ≤ position returned ≤ board.size()
   */
  private int getIndexForFit(NumberTile tile)
  {  /* to be implemented in part (a) */  }

  /** Places tile on the game board if it fits (checking all possible tile orientations if necessary).
   *  If there are no tiles on the game board, the tile is placed at position 0.
   *  The tile should be placed at most 1 time.
   *  Precondition: board is not null
   *  @param tile the tile to be placed on the game board
   *  @return true if tile is placed successfully; false otherwise
   *  Postcondition: the orientations of the other tiles on the board are not changed
   *  Postcondition: the order of the other tiles on the board relative to each other is not changed
   */
  public boolean insertTile(NumberTile tile)
  {  /* to be implemented in part (b) */  }

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

(a) Write the `TileGame` method `getIndexForFit` that determines where a given tile, in its current orientation, fits on the game board. A tile can be inserted at either end of a game board or between two existing tiles if the side(s) of the new tile match the adjacent side(s) of the tile(s) currently on the game board. If there are no tiles on the game board, the position for the insert is 0. The method returns the position that the new tile will occupy on the game board after it has been inserted. If there are multiple possible positions for the tile, the method will return any one of them. If the given tile does not fit anywhere on the game board, the method returns `-1`.

For example, the following diagram shows a game board and two potential tiles to be placed. The call `getIndexForFit(tile1)` can return either 3 or 4 because `tile1` can be inserted between the tiles at positions 2 and 3, or between the tiles at positions 3 and 4. The call `getIndexForFit(tile2)` returns `-1` because `tile2`, in its current orientation, does not fit anywhere on the game board.



Complete method `getIndexForFit` below.

```
/** Determines where to insert tile, in its current orientation, into game board
 *  @param tile the tile to be placed on the game board
 *  @return the position of tile where tile is to be inserted:
 *            0 if the board is empty;
 *           -1 if tile does not fit in front, at end, or between any existing tiles;
 *           otherwise, 0 ≤ position returned ≤ board.size()
 */
private int getIndexForFit(NumberTile tile)
```

(b) Write the `TileGame` method `insertTile` that attempts to insert the given tile on the game board. The method returns `true` if the tile is inserted successfully and `false` only if the tile cannot be placed on the board in any orientation.

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

Complete method `insertTile` below.

```
/** Places tile on the game board if it fits (checking all possible tile orientations if necessary).
 *  If there are no tiles on the game board, the tile is placed at position 0.
 *  The tile should be placed at most 1 time.
 *  Precondition: board is not null
 *  @param tile the tile to be placed on the game board
 *  @return true if tile is placed successfully; false otherwise
 *  Postcondition: the orientations of the other tiles on the board are not changed
 *  Postcondition: the order of the other tiles on the board relative to each other is not changed
 */
public boolean insertTile(NumberTile tile)
```

STOP

END OF EXAM

10. 2019(1). 3

10. A student plans to analyze product reviews found on a Web site by looking for keywords in posted reviews. The `ProductReview` class, shown below, is used to represent a single review. A product review consists of a product name and a review of that product.

```
public class ProductReview
{
   private String name;
   private String review;

   /** Constructs a ProductReview object and initializes the instance variables. */
   public ProductReview(String pName, String pReview)
   {
      name = pName;
      review = pReview;
   }

   /** Returns the name of the product. */
   public String getName()
   {  return name; }

   /** Returns the review of the product. */
   public String getReview()
   {  return review; }
}
```

The `ReviewCollector` class, shown below, is used to represent a collection of reviews to be analyzed.

```
public class ReviewCollector
{
   private ArrayList<ProductReview> reviewList;
   private ArrayList<String> productList;

   /** Constructs a ReviewCollector object and initializes the instance variables. */
   public ReviewCollector()
   {
      reviewList = new ArrayList<ProductReview>();
      productList = new ArrayList<String>();
   }

   /** Adds a new review to the collection of reviews, as described in part (a). */
   public void addReview(ProductReview prodReview)
   {  /* to be implemented in part (a) */  }

   /** Returns the number of good reviews for a given product name, as described in part (b). */
   public int getNumGoodReviews(String prodName)
   {  /* to be implemented in part (b) */  }

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

(a) Write the `addReview` method, which adds a single product review, represented by a `ProductReview` object, to the `ReviewCollector` object. The `addReview` method does the following when it adds a product review.

- The `ProductReview` object is added to the `reviewList` instance variable.
- The product name from the `ProductReview` object is added to the `productList` instance variable if the product name is not already found in `productList`.

Elements may be added to `reviewList` and `productList` in any order.

Complete method `addReview`.

```
/** Adds a new review to the collection of reviews, as described in part (a). */
public void addReview(ProductReview prodReview)
```

(b) Write the `getNumGoodReviews` method, which returns the number of *good* reviews for a given product name. A review is considered good if it contains the string `"best"` (all lowercase). If there are no reviews with a matching product name, the method returns `0`. Note that a review that contains `"BEST"` or `"Best"` is not considered a good review (since not all the letters of `"best"` are lowercase), but a review that contains `"asbestos"` is considered a good review (since all the letters of `"best"` are lowercase).

Complete method `getNumGoodReviews`.

```
/** Returns the number of good reviews for a given product name, as described in part (b). */
public int getNumGoodReviews(String prodName)
```

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 ProductReview

private String name
private String review

public ProductReview(String pName, String pReview)
public String getName()
public String getReview()

public class ReviewCollector

private ArrayList<ProductReview> reviewList
private ArrayList<String> productList

public ReviewCollector()
public void addReview(ProductReview prodReview)
public int getNumGoodReviews(String prodName)
```

11. 2007.3

11. Consider a system for processing student test scores. The following class will be used as part of this system and contains a student's name and the student's answers for a multiple-choice test. The answers are represented as strings of length one with an omitted answer being represented by a string containing a single question mark ("?"). These answers are stored in an `ArrayList` in which the position of the answer corresponds to the question number on the test (question numbers start at 0). A student's score on the test is computed by comparing the student's answers with the corresponding answers in the answer key for the test. One point is awarded for each correct answer and ¼ of a point is deducted for each incorrect answer. Omitted answers (indicated by "?") do not change the student's score.

```
public class StudentAnswerSheet
{
  private ArrayList<String> answers;   // the list of the student's answers


  /** @param key  the list of correct answers, represented as strings of length one
   *             Precondition: key.size() is equal to the number of answers in this answer sheet
   *  @return  this student's test score
   */
  public double getScore(ArrayList<String> key)
  {   /* to be implemented in part (a) */   }


  /** @return  the name of the student
   */
  public String getName()
  {   /* implementation not shown */   }

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

The following table shows an example of an answer key, a student's answers, and the corresponding point values that would be awarded for the student's answers. In this example, there are six correct answers, three incorrect answers, and one omitted answer. The student's score is ((6 * 1) - (3 * 0.25)) = 5.25 .

Question number	0	1	2	3	4	5	6	7	8	9
`key`	"A"	"C"	"D"	"E"	"B"	"C"	"E"	"B"	"B"	"C"
`answers`	"A"	"B"	"D"	"E"	"A"	"C"	"?"	"B"	"D"	"C"
Points awarded	1	−0.25	1	1	−0.25	1	0	1	−0.25	1

(a) Write the `StudentAnswerSheet` method `getScore`. The parameter passed to method `getScore` is an `ArrayList` of strings representing the correct answer key for the test being scored. The method computes and returns a `double` that represents the score for the student's test answers when compared with the answer key. One point is awarded for each correct answer and ¼ of a point is deducted for each incorrect answer. Omitted answers (indicated by `"?"`) do not change the student's score.

Complete method `getScore` below.

```
/** @param key  the list of correct answers, represented as strings of length one
 *             Precondition:  key.size()  is equal to the number of answers in this answer sheet
 *  @return  this student's test score
 */
public double getScore(ArrayList<String> key)
```

(b) Consider the following class that represents the test results of a group of students that took a multiple-choice test.

```
public class TestResults
{
  private ArrayList<StudentAnswerSheet> sheets;

  /** Precondition: sheets.size() > 0;
   *                all answer sheets in sheets have the same number of answers
   *  @param key the list of correct answers represented as strings of length one
   *             Precondition: key.size() is equal to the number of answers
   *                           in each of the answer sheets in sheets
   *  @return the name of the student with the highest score
   */
  public String highestScoringStudent(ArrayList<String> key)
  {  /* to be implemented in part (b) */  }

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

Write the `TestResults` method `highestScoringStudent`, which returns the name of the student who received the highest score on the test represented by the parameter `key`. If there is more than one student with the highest score, the name of any one of these highest-scoring students may be returned. You may assume that the size of each answer sheet represented in the `ArrayList sheets` is equal to the size of the `ArrayList key`.

In writing `highestScoringStudent`, assume that `getScore` works as specified, regardless of what you wrote in part (a).

Complete method `highestScoringStudent` below.

```
  /** Precondition: sheets.size() > 0;
   *                all answer sheets in sheets have the same number of answers
   *  @param key the list of correct answers represented as strings of length one
   *             Precondition: key.size() is equal to the number of answers
   *                           in each of the answer sheets in sheets
   *  @return the name of the student with the highest score
   */
  public String highestScoringStudent(ArrayList<String> key)
```

12. 2008.2

12. Consider a method of encoding and decoding words that is based on a *master string*. This master string will contain all the letters of the alphabet, some possibly more than once. An example of a master string is "sixtyzipperswerequicklypickedfromthewovenjutebag". This string and its indexes are shown below.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
s	i	x	t	y	z	i	p	p	e	r	s	w	e	r	e	q	u	i	c	k	l	y	p

24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
i	c	k	e	d	f	r	o	m	t	h	e	w	o	v	e	n	j	u	t	e	b	a	g

An encoded string is defined by a list of *string parts*. A string part is defined by its starting index in the master string and its length. For example, the string "overeager" is encoded as the list of string parts [(37, 3), (14, 2), (46, 2), (9, 2)] denoting the substrings "ove", "re", "ag", and "er".

String parts will be represented by the StringPart class shown below.

```
public class StringPart
{
  /** @param start  the starting position of the substring in a master string
   *  @param length  the length of the substring in a master string
   */
  public StringPart(int start, int length)
  {  /* implementation not shown */  }


  /** @return  the starting position of the substring in a master string
   */
  public int getStart()
  {  /* implementation not shown */  }


  /** @return  the length of the substring in a master string
   */
  public int getLength()
  {  /* implementation not shown */  }

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

The class StringCoder provides methods to encode and decode words using a given master string. When encoding, there may be multiple matching string parts of the master string. The helper method findPart is provided to choose a string part within the master string that matches the beginning of a given string.

```
public class StringCoder
{
  private String masterString;

  /** @param master  the master string for the StringCoder
   *              Precondition: the master string contains all the letters of the alphabet
   */
  public StringCoder(String master)
  {  masterString = master;  }


  /** @param parts  an ArrayList of string parts that are valid in the master string
   *              Precondition: parts.size() > 0
   *  @return  the string obtained by concatenating the parts of the master string
   */
  public String decodeString(ArrayList<StringPart> parts)
  {  /* to be implemented in part (a) */  }


  /** @param str  the string to encode using the master string
   *              Precondition: all of the characters in str appear in the master string;
   *                            str.length() > 0
   *  @return  a string part in the master string that matches the beginning of str.
   *               The returned string part has length at least 1.
   */
  private StringPart findPart(String str)
  {  /* implementation not shown */  }


  /** @param word  the string to be encoded
   *              Precondition: all of the characters in word appear in the master string;
   *                            word.length() > 0
   *  @return  an ArrayList of string parts of the master string that can be combined
   *               to create word
   */
  public ArrayList<StringPart> encodeString(String word)
  {  /* to be implemented in part (b) */  }

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

(a) Write the StringCoder method decodeString. This method retrieves the substrings in the master string represented by each of the StringPart objects in parts, concatenates them in the order in which they appear in parts, and returns the result.

Complete method decodeString below.

```
/** @param parts an ArrayList of string parts that are valid in the master string
 *          Precondition: parts.size() > 0
 *  @return the string obtained by concatenating the parts of the master string
 */
public String decodeString(ArrayList<StringPart> parts)
```

(b) Write the StringCoder method encodeString. A string is encoded by determining the substrings in the master string that can be combined to generate the given string. The encoding starts with a string part that matches the beginning of the word, followed by a string part that matches the beginning of the rest of the word, and so on. The string parts are returned in an array list in the order in which they appear in word.

The helper method findPart must be used to choose matching string parts in the master string.

Complete method encodeString below.

```
/** @param word  the string to be encoded
 *           Precondition: all of the characters in word appear in the master string;
 *                        word.length() > 0
 *  @return an ArrayList of string parts of the master string that can be combined
 *           to create word
 */
public ArrayList<StringPart> encodeString(String word)
```

13. 2016.4

13. 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)
```

(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)
```

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)
```

(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)
```

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)
```

14. 2016.2

14. 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)
```

(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"`

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)
```

(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)
```

15. 2006.1

15. An appointment scheduling system is represented by the following three classes: TimeInterval, Appointment, and DailySchedule. In this question, you will implement one method in the Appointment class and two methods in the DailySchedule class.

A TimeInterval object represents a period of time. The TimeInterval class provides a method to determine if another time interval overlaps with the time interval represented by the current TimeInterval object. An Appointment object contains a time interval for the appointment and a method that determines if there is a time conflict between the current appointment and another appointment. The declarations of the TimeInterval and Appointment classes are shown below.

```
public class TimeInterval
{
  // returns true if interval overlaps with this TimeInterval;
  // otherwise, returns false
  public boolean overlapsWith(TimeInterval interval)
  {  /* implementation not shown */  }

  // There may be fields, constructors, and methods that are not shown.
}


public class Appointment
{
  // returns the time interval of this Appointment
  public TimeInterval getTime()
  {  /* implementation not shown */  }

  // returns true if the time interval of this Appointment
  // overlaps with the time interval of other;
  // otherwise, returns false
  public boolean conflictsWith(Appointment other)
  {  /* to be implemented in part (a) */  }

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

(a) Write the Appointment method conflictsWith. If the time interval of the current appointment overlaps with the time interval of the appointment other, method conflictsWith should return true, otherwise, it should return false.

Complete method conflictsWith below.

```
  // returns true if the time interval of this Appointment
  // overlaps with the time interval of other;
  // otherwise, returns false
  public boolean conflictsWith(Appointment other)
```

(b) A DailySchedule object contains a list of nonoverlapping Appointment objects. The DailySchedule class contains methods to clear all appointments that conflict with a given appointment and to add an appointment to the schedule.

```
public class DailySchedule
{
  // contains Appointment objects, no two Appointments overlap
  private ArrayList apptList;


  public DailySchedule()
  {  apptList = new ArrayList();  }


  // removes all appointments that overlap the given Appointment
  // postcondition: all appointments that have a time conflict with
  //                appt have been removed from this DailySchedule
  public void clearConflicts(Appointment appt)
  {  /* to be implemented in part (b) */  }


  // if emergency is true, clears any overlapping appointments and adds
  // appt to this DailySchedule; otherwise, if there are no conflicting
  // appointments, adds appt to this DailySchedule;
  // returns true if the appointment was added;
  // otherwise, returns false
  public boolean addAppt(Appointment appt, boolean emergency)
  {  /* to be implemented in part (c) */  }


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

Write the DailySchedule method clearConflicts. Method clearConflicts removes all appointments that conflict with the given appointment.

In writing method clearConflicts, you may assume that conflictsWith works as specified, regardless of what you wrote in part (a).

Complete method clearConflicts below.

```
  // removes all appointments that overlap the given Appointment
  // postcondition: all appointments that have a time conflict with
  //                appt have been removed from this DailySchedule
  public void clearConflicts(Appointment appt)
```

(c) Write the DailySchedule method addAppt. The parameters to method addAppt are an appointment and a boolean value that indicates whether the appointment to be added is an emergency. If the appointment is an emergency, the schedule is cleared of all appointments that have a time conflict with the given appointment and the appointment is added to the schedule. If the appointment is not an emergency, the schedule is checked for any conflicting appointments. If there are no conflicting appointments, the given appointment is added to the schedule. Method addAppt returns true if the appointment was added to the schedule; otherwise, it returns false.

In writing method addAppt, you may assume that conflictsWith and clearConflicts work as specified, regardless of what you wrote in parts (a) and (b).

Complete method addAppt below.

```
// if emergency is true, clears any overlapping appointments and adds
// appt to this DailySchedule; otherwise, if there are no conflicting
// appointments, adds appt to this DailySchedule;
// returns true if the appointment was added;
// otherwise, returns false
public boolean addAppt(Appointment appt, boolean emergency)
```

16. 2015.3

16 A two-dimensional array of integers in which most elements are zero is called a *sparse array*. Because most elements have a value of zero, memory can be saved by storing only the non-zero values along with their row and column indexes. The following complete `SparseArrayEntry` class is used to represent non-zero elements in a sparse array. A `SparseArrayEntry` object cannot be modified after it has been constructed.

```
public class SparseArrayEntry
{
  /** The row index and column index for this entry in the sparse array */
  private int row;
  private int col;

  /** The value of this entry in the sparse array */
  private int value;

  /** Constructs a SparseArrayEntry object that represents a sparse array element
   *   with row index r and column index c, containing value v.
   */
  public SparseArrayEntry(int r, int c, int v)
  {
    row = r;
    col = c;
    value = v;
  }

  /** Returns the row index of this sparse array element. */
  public int getRow()
  {  return row;  }

  /** Returns the column index of this sparse array element. */
  public int getCol()
  {  return col;  }

  /** Returns the value of this sparse array element. */
  public int getValue()
  {  return value;  }
}
```

The SparseArray class represents a sparse array. It contains a list of SparseArrayEntry objects, each of which represents one of the non-zero elements in the array. The entries representing the non-zero elements are stored in the list in no particular order. Each non-zero element is represented by exactly one entry in the list.

```
public class SparseArray
{
   /** The number of rows and columns in the sparse array. */
   private int numRows;
   private int numCols;

   /** The list of entries representing the non-zero elements of the sparse array. Entries are stored in the
    *  list in no particular order. Each non-zero element is represented by exactly one entry in the list.
    */
   private List<SparseArrayEntry> entries;

   /** Constructs an empty SparseArray. */
   public SparseArray()
   {  entries = new ArrayList<SparseArrayEntry>();  }

   /** Returns the number of rows in the sparse array. */
   public int getNumRows()
   {  return numRows;  }

   /** Returns the number of columns in the sparse array. */
   public int getNumCols()
   {  return numCols;  }

   /** Returns the value of the element at row index row and column index col in the sparse array.
    *  Precondition: 0 ≤ row < getNumRows()
    *                0 ≤ col < getNumCols()
    */
   public int getValueAt(int row, int col)
   {  /* to be implemented in part (a) */  }

   /** Removes the column col from the sparse array.
    *  Precondition: 0 ≤ col < getNumCols()
    */
   public void removeColumn(int col)
   {  /* to be implemented in part (b) */  }

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

The following table shows an example of a two-dimensional sparse array. Empty cells in the table indicate zero values.

	0	1	2	3	4
0					
1		5			4
2	1				
3		-9			
4					
5					

The sample array can be represented by a `SparseArray` object, `sparse`, with the following instance variable values. The items in `entries` are in no particular order; one possible ordering is shown below.

```
numRows: 6

numCols: 5
```

entries:				
	row: 1 col: 4 value: 4	row: 2 col: 0 value: 1	row: 3 col: 1 value: -9	row: 1 col: 1 value: 5

(a) Write the `SparseArray` method `getValueAt`. The method returns the value of the sparse array element at a given row and column in the sparse array. If the list `entries` contains an entry with the specified row and column, the value associated with the entry is returned. If there is no entry in `entries` corresponding to the specified row and column, 0 is returned.

In the example above, the call `sparse.getValueAt(3, 1)` would return -9, and `sparse.getValueAt(3, 3)` would return 0.

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Complete method getValueAt below.

```
/** Returns the value of the element at row index row and column index col in the sparse array.
 *  Precondition: 0 ≤ row < getNumRows()
 *                0 ≤ col < getNumCols()
 */
public int getValueAt(int row, int col)
```

(b) Write the `SparseArray` method `removeColumn`. After removing a specified column from a sparse array:

- All entries in the list `entries` with column indexes matching `col` are removed from the list.
- All entries in the list `entries` with column indexes greater than `col` are replaced by entries with column indexes that are decremented by one (moved one column to the left).
- The number of columns in the sparse array is adjusted to reflect the column removed.

The sample object `sparse` from the beginning of the question is repeated for your convenience.

	0	1	2	3	4
0					
1		5			4
2	1				
3		-9			
4					
5					

The shaded entries in `entries`, below, correspond to the shaded column above.

```
numRows: 6

numCols: 5
```

entries:	row: 1 col: 4 value: 4	row: 2 col: 0 value: 1	row: 3 col: 1 value: -9	row: 1 col: 1 value: 5

When `sparse` has the state shown above, the call `sparse.removeColumn(1)` could result in `sparse` having the following values in its instance variables (since `entries` is in no particular order, it would be equally valid to reverse the order of its two items). The shaded areas below show the changes.

```
numRows: 6

numCols: 4
```

entries:	row: 1 col: 3 value: 4	row: 2 col: 0 value: 1

Class information repeated from the beginning of the question

```
public class SparseArrayEntry

public SparseArrayEntry(int r, int c, int v)
public int getRow()
public int getCol()
public int getValue()

public class SparseArray

private int numRows
private int numCols
private List<SparseArrayEntry> entries
public int getNumRows()
public int getNumCols()
public int getValueAt(int row, int col)
public void removeColumn(int col)
```

Complete method `removeColumn` below.

```
/** Removes the column col from the sparse array.
 *  Precondition: 0 ≤ col < getNumCols()
 */
public void removeColumn(int col)
```

17. 2014.1

17. This question involves reasoning about strings made up of uppercase letters. You will implement two related methods that appear in the same class (not shown). The first method takes a single string parameter and returns a scrambled version of that string. The second method takes a list of strings and modifies the list by scrambling each entry in the list. Any entry that cannot be scrambled is removed from the list.

(a) Write the method `scrambleWord`, which takes a given word and returns a string that contains a scrambled version of the word according to the following rules.

- The scrambling process begins at the first letter of the word and continues from left to right.
- If two consecutive letters consist of an "A" followed by a letter that is not an "A", then the two letters are swapped in the resulting string.
- Once the letters in two adjacent positions have been swapped, neither of those two positions can be involved in a future swap.

The following table shows several examples of words and their scrambled versions.

`word`	Result returned by `scrambleWord(word)`
"TAN"	"TNA"
"ABRACADABRA"	"BARCADABARA"
"WHOA"	"WHOA"
"AARDVARK"	"ARADVRAK"
"EGGS"	"EGGS"
"A"	"A"
""	""

Complete method `scrambleWord` below.

```
/** Scrambles a given word.
 *  @param word the word to be scrambled
 *  @return the scrambled word (possibly equal to word)
 *  Precondition: word is either an empty string or contains only uppercase letters.
 *  Postcondition: the string returned was created from word as follows:
 *     – the word was scrambled, beginning at the first letter and continuing from left to right
 *     – two consecutive letters consisting of "A" followed by a letter that was not "A" were swapped
 *     – letters were swapped at most once
 */
public static String scrambleWord(String word)
```

(b) Write the method `scrambleOrRemove`, which replaces each word in the parameter `wordList` with its scrambled version and removes any words that are unchanged after scrambling. The relative ordering of the entries in `wordList` remains the same as before the call to `scrambleOrRemove`.

The following example shows how the contents of `wordList` would be modified as a result of calling `scrambleOrRemove`.

Before the call to `scrambleOrRemove`:

	0	1	2	3	4
`wordList`	"TAN"	"ABRACADABRA"	"WHOA"	"APPLE"	"EGGS"

After the call to `scrambleOrRemove`:

	0	1	2
`wordList`	"TNA"	"BARCADABARA"	"PAPLE"

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Assume that `scrambleWord` is in the same class as `scrambleOrRemove` and works as specified, regardless of what you wrote in part (a).

Complete method `scrambleOrRemove` below.

```
/** Modifies wordList by replacing each word with its scrambled
 *  version, removing any words that are unchanged as a result of scrambling.
 *  @param wordList the list of words
 *  Precondition: wordList contains only non-null objects
 *  Postcondition:
 *    - all words unchanged by scrambling have been removed from wordList
 *    - each of the remaining words has been replaced by its scrambled version
 *    - the relative ordering of the entries in wordList is the same as it was
 *        before the method was called
 */
public static void scrambleOrRemove(List<String> wordList)
```

18. 2011.3

18. A fuel depot has a number of fuel tanks arranged in a line and a robot that moves a filling mechanism back and forth along the line so that the tanks can be filled. A fuel tank is specified by the `FuelTank` class below.

```
public class FuelTank  {

   /** @return an integer value that ranges from  0 (empty) to 100 (full)  */
   int getFuelLevel();
}
```

A fuel depot keeps track of the fuel tanks and the robot. The following figure represents the tanks and the robot in a fuel depot. The robot, indicated by the arrow, is currently at index 2 and is facing to the right.

Tank index	0	1	2	3	4	5
Fuel level in tank	80	70	20	45	50	25
Robot			➔			

The state of the robot includes the index of its location and the direction in which it is facing (to the right or to the left). This information is specified in the `FuelRobot` class as shown in the following declaration.

```
public class FuelRobot  {

   /** @return the index of the current location of the robot  */
   int getCurrentIndex();


   /**  Determine whether the robot is currently facing to the right
    *   @return true   if the robot is facing to the right (toward tanks with larger indexes)
    *           false  if the robot is facing to the left (toward tanks with smaller indexes)
    */
   boolean isFacingRight();


   /**  Changes the current direction of the robot  */
   void changeDirection();


   /**  Moves the robot in its current direction by the number of locations specified.
    *   @param numLocs  the number of locations to move. A value of 1 moves
    *                   the robot to the next location in the current direction.
    *             Precondition: numLocs > 0
    */
   void moveForward(int numLocs);
}
```

A fuel depot is represented by the FuelDepot class as shown in the following class declaration.

```
public class FuelDepot
{
   /** The robot used to move the filling mechanism */
   private FuelRobot filler;

   /** The list of fuel tanks */
   private List<FuelTank> tanks;

   /** Determines and returns the index of the next tank to be filled.
    *  @param threshold fuel tanks with a fuel level ≤ threshold may be filled
    *  @return index of the location of the next tank to be filled
    *  Postcondition: the state of the robot has not changed
    */
   public int nextTankToFill(int threshold)
   {  /* to be implemented in part (a) */  }


   /** Moves the robot to location locIndex.
    *  @param locIndex the index of the location of the tank to move to
    *         Precondition: 0 ≤ locIndex < tanks.size()
    *  Postcondition: the current location of the robot is locIndex
    */
   public void moveToLocation(int locIndex)
   {  /* to be implemented in part (b) */  }

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

(a) Write the `FuelDepot` method `nextTankToFill` that returns the index of the next tank to be filled.

The index for the next tank to be filled is determined according to the following rules:

- Return the index of a tank with the lowest fuel level that is less than or equal to a given threshold. If there is more than one fuel tank with the same lowest fuel level, any of their indexes can be returned.
- If there are no tanks with a fuel level less than or equal to the threshold, return the robot's current index.

For example, suppose the tanks contain the fuel levels shown in the following figure.

Tank index	0	1	2	3	4	5	6
Fuel level in tank	20	30	80	55	50	75	20
Robot			➔				

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

`threshold`	Return Value	Rationale
50	0 or 6	20 is the lowest fuel level, so either 0 or 6 can be returned.
15	2	There are no tanks with a fuel level ≤ `threshold`, so the robot's current index is returned.

Complete method `nextTankToFill` below.

```
/** Determines and returns the index of the next tank to be filled.
 *  @param threshold fuel tanks with a fuel level ≤ threshold may be filled
 *  @return index of the location of the next tank to be filled
 *  Postcondition: the state of the robot has not changed
 */
public int nextTankToFill(int threshold)
```

(b) Write the `FuelDepot` method `moveToLocation` that will move the robot to the given tank location. Because the robot can only move forward, it may be necessary to change the direction of the robot before having it move. Do not move the robot past the end of the line of fuel tanks.

Complete method `moveToLocation` below.

```
/** Moves the robot to location locIndex.
 *  @param locIndex the index of the location of the tank to move to
 *          Precondition: 0 ≤ locIndex < tanks.size()
 *  Postcondition: the current location of the robot is locIndex
 */
public void moveToLocation(int locIndex)
```

19. 2008.1

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

10. A travel agency maintains a list of information about airline flights. Flight information includes a departure time and an arrival time. You may assume that the two times occur on the same day. These times are represented by objects of the `Time` class.

The declaration for the `Time` class is shown below. It includes a method `minutesUntil` that returns the difference (in minutes) between the current `Time` object and another `Time` object.

```
public class Time
{
  /** @return  difference, in minutes, between this time and other;
   *           difference is negative if other is earlier than this time
   */
  public int minutesUntil(Time other)
  {  /* implementation not shown */  }

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

For example, assume that `t1` and `t2` are `Time` objects where `t1` represents 1:00 P.M. and `t2` represents 2:15 P.M. The call `t1.minutesUntil(t2)` will return `75` and the call `t2.minutesUntil(t1)` will return `-75`.

The declaration for the `Flight` class is shown below. It has methods to access the departure time and the arrival time of a flight. You may assume that the departure time of a flight is earlier than its arrival time.

```
public class Flight
{
   /** @return  time at which the flight departs
    */
   public Time getDepartureTime()
   { /*  implementation not shown  */ }

   /** @return  time at which the flight arrives
    */
   public Time getArrivalTime()
   { /*  implementation not shown  */ }

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

A trip consists of a sequence of flights and is represented by the `Trip` class. The `Trip` class contains an `ArrayList` of `Flight` objects that are stored in chronological order. You may assume that for each flight after the first flight in the list, the departure time of the flight is later than the arrival time of the preceding flight in the list. A partial declaration of the `Trip` class is shown below. You will write two methods for the `Trip` class.

```
public class Trip
{
  private ArrayList<Flight> flights;
     // stores the flights (if any) in chronological order

  /** @return the number of minutes from the departure of the first flight to the arrival
   *          of the last flight if there are one or more flights in the trip;
   *          0, if there are no flights in the trip
   */
  public int getDuration()
  {  /* to be implemented in part (a) */  }

  /** Precondition: the departure time for each flight is later than the arrival time of its
   *                preceding flight
   *  @return the smallest number of minutes between the arrival of a flight and the departure
   *          of the flight immediately after it, if there are two or more flights in the trip;
   *          -1, if there are fewer than two flights in the trip
   */
  public int getShortestLayover()
  {  /* to be implemented in part (b) */  }

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

(a) Complete method `getDuration` below.

```
/** @return  the number of minutes from the departure of the first flight to the arrival
 *           of the last flight if there are one or more flights in the trip;
 *           0, if there are no flights in the trip
 */
public int getDuration()
```

(b) Write the `Trip` method `getShortestLayover`. A layover is the number of minutes from the arrival of one flight in a trip to the departure of the flight immediately after it. If there are two or more flights in the trip, the method should return the shortest layover of the trip; otherwise, it should return -1.

For example, assume that the instance variable `flights` of a `Trip` object `vacation` contains the following flight information.

	Departure Time	**Arrival Time**	**Layover (minutes)**
Flight 0	11:30 a.m.	12:15 p.m.	
			} 60
Flight 1	1:15 p.m.	3:45 p.m.	
			} 15
Flight 2	4:00 p.m.	6:45 p.m.	
			} 210
Flight 3	10:15 p.m.	11:00 p.m.	

The call `vacation.getShortestLayover()` should return 15.

Complete method `getShortestLayover` below.

```
/** Precondition: the departure time for each flight is later than the arrival time of its
 *                preceding flight
 *  @return  the smallest number of minutes between the arrival of a flight and the departure
 *           of the flight immediately after it, if there are two or more flights in the trip;
 *           -1, if there are fewer than two flights in the trip
 */
public int getShortestLayover()
```

Apply the question assessment rubric first, which always takes precedence. Penalty points can only be deducted in a part of the question that has earned credit via the question rubric. No part of a question (a, b, c) may have a negative point total. A given penalty can be assessed only once for a question, even if it occurs multiple times or in multiple parts of that question. A maximum of 3 penalty points may be assessed per question.

1-Point Penalty

v) Array/collection access confusion (`[] get`)

w) Extraneous code that causes side-effect (e.g., writing to output, failure to compile)

x) Local variables used but none declared

y) Destruction of persistent data (e.g., changing value referenced by parameter)

z) Void method or constructor that returns a value

No Penalty

- Extraneous code with no side-effect (e.g., precondition check, no-op)
- Spelling/case discrepancies where there is no ambiguity*
- Local variable not declared provided other variables are declared in some part
- `private` or `public` qualifier on a local variable
- Missing `public` qualifier on class or constructor header
- Keyword used as an identifier
- Common mathematical symbols used for operators **(× • ÷ ≤ ≥ <> ≠)**
- `[]` vs. `()` vs. `<>`
- = instead of == and vice versa
- `length`/`size` confusion for array, String, List, or ArrayList; with or without ( )
- Extraneous `[]` when referencing entire array
- `[i,j]` instead of `[i][j]`
- Extraneous size in array declaration, e.g., `int[size] nums = new int[size];`
- Missing ; where structure clearly conveys intent
- Missing { } where indentation clearly conveys intent
- Missing ( ) on parameter-less method or constructor invocations
- Missing ( ) around `if` or `while` conditions

Spelling and case discrepancies for identifiers fall under the "No Penalty" category only if the correction can be* ***unambiguously* *inferred from context. For example,* "ArayList" *instead of* "ArrayList". *As a counter example, note that if the code declares* "Bug bug;", *then uses* "Bug.move()" *instead of* "bug.move()", *the context does* ***not*** *allow for the reader to assume the object instead of the class.*

Basic Part-1 Multiple Choice

1. Consider the following code segment.

```
int a = 3 + 2 * 3;
int b = 4 + 3 / 2;
int c = 7 % 4 + 3;
double d = a + b + c;
```

What is the value of `d` after the code segment is executed?

(A) `14.0`

(B) `18.0`

(C) `20.0`

(D) `20.5`

(E) `26.0`

---

2. Consider the following code segment. Assume `num` is a properly declared and initialized `int` variable.

```
if (num > 0)
{
   if (num % 2 == 0)
   {
      System.out.println("A");
   }
   else
   {
      System.out.println("B");
   }
}
```

Which of the following best describes the result of executing the code segment?

(A) When `num` is a negative odd integer, `"B"` is printed; otherwise, `"A"` is printed.

(B) When `num` is a negative even integer, `"B"` is printed; otherwise, nothing is printed.

(C) When `num` is a positive even integer, `"A"` is printed; otherwise, `"B"` is printed.

(D) When `num` is a positive even integer, `"A"` is printed; when `num` is a positive odd integer, `"B"` is printed; otherwise, nothing is printed.

(E) When `num` is a positive odd integer, `"A"` is printed; when `num` is a positive even integer, `"B"` is printed; otherwise, nothing is printed.

3. Which of the following code segments produces the output `"987654321"` ?

(A)
```
int num = 10;
while (num > 0)
{
   System.out.print(num);
   num--;
}
```

(B)
```
int num = 10;
while (num >= 0)
{
   System.out.print(num);
   num--;
}
```

(C)
```
int num = 10;
while (num > 1)
{
   num--;
   System.out.print(num);
}
```

(D)
```
int num = 10;
while (num >= 1)
{
   num--;
   System.out.print(num);
}
```

(E)
```
int num = 0;
while (num <= 9)
{
   System.out.print(10 - num);
   num++;
}
```

4. Which of the following expressions evaluate to `3.5` ?

I. `(double) 2 / 4 + 3`

II. `(double) (2 / 4) + 3`

III. `(double) (2 / 4 + 3)`

A) I only

B) III only

C) I and II only

D) II and III only

E) I, II, and III

5. Consider the following code segment.

```
int num = /* initial value not shown */;
boolean b1 = true;
if (num > 0)
{
   if (num >= 100)
   {
      b1 = false;
   }
}
else
{
   if (num >= -100)
   {
      b1 = false;
   }
}
```

Which of the following statements assigns the same value to `b2` as the code segment assigns to `b1` for all values of `num` ?

(A) `boolean b2 = (num > -100) && (num < 100);`

(B) `boolean b2 = (num > -100) || (num < 100);`

(C) `boolean b2 = (num < -100) || (num > 100);`

(D) `boolean b2 = (num < -100) && (num > 0 || num < 100);`

(E) `boolean b2 = (num < -100) || (num > 0 && num < 100);`

6. Assume that `a`, `b`, and `c` are `boolean` variables that have been properly declared and initialized. Which of the following `boolean` expressions is equivalent to `!(a && b) || c` ?

(A) `a && b  && c`

(B) `a || b || c`

(C) `!a && !b || c`

(D) `!a && !b && c`

(E) `!a || !b || c`

7. Consider the following two code segments. Assume that the `int` variables `m` and `n` have been properly declared and initialized and are both greater than `0`.

```
 I. for (int i = 0; i < m * n; i++)
    {
       System.out.print("A");
    }

II. for (int j = 1; j <= m; j++)
    {
       for (int k = 1; k < n; k++)
       {
          System.out.print("B");
       }
    }
```

Assume that the initial values of `m` and `n` are the same in code segment I as they are in code segment II. Which of the following correctly compares the number of times that `"A"` and `"B"` are printed when each code segment is executed?

(A) `"A"` is printed `m` fewer times than `"B"`.

(B) `"A"` is printed `n` fewer times than `"B"`.

(C) `"A"` is printed `m` more times than `"B"`.

(D) `"A"` is printed `n` more times than `"B"`.

(E) `"A"` and `"B"` are printed the same number of times.

8. Consider the following statement. Assume that `a` and `b` are properly declared and initialized `boolean` variables.

```
boolean c = (a && b) || (!a && b);
```

Under which of the following conditions will `c` be assigned the value `false` ?

(A) Always

(B) Never

(C) When `a` and `b` have the same value

(D) When `a` has the value `false`

(E) When `b` has the value `false`

9. Consider the following code segment. Assume that `num3 > num2 > 0`.

```
int num1 = 0;
int num2 = /* initial value not shown */;
int num3 = /* initial value not shown */;

while (num2 < num3)
{
   num1 += num2;
   num2++;
}
```

Which of the following best describes the contents of `num1` as a result of executing the code segment?

(A) The product of `num2` and `num3`

(B) The product of `num2` and `num3 - 1`

(C) The sum of `num2` and `num3`

(D) The sum of all integers from `num2` to `num3`, inclusive

(E) The sum of all integers from `num2` to `num3 - 1`, inclusive

10. Consider the following code segment.

```
int x = 7;
int y = 3;

if ((x < 10) && (y < 0))
  System.out.println("Value is: " + x * y);
else
  System.out.println("Value is: " + x / y);
```

What is printed as a result of executing the code segment?

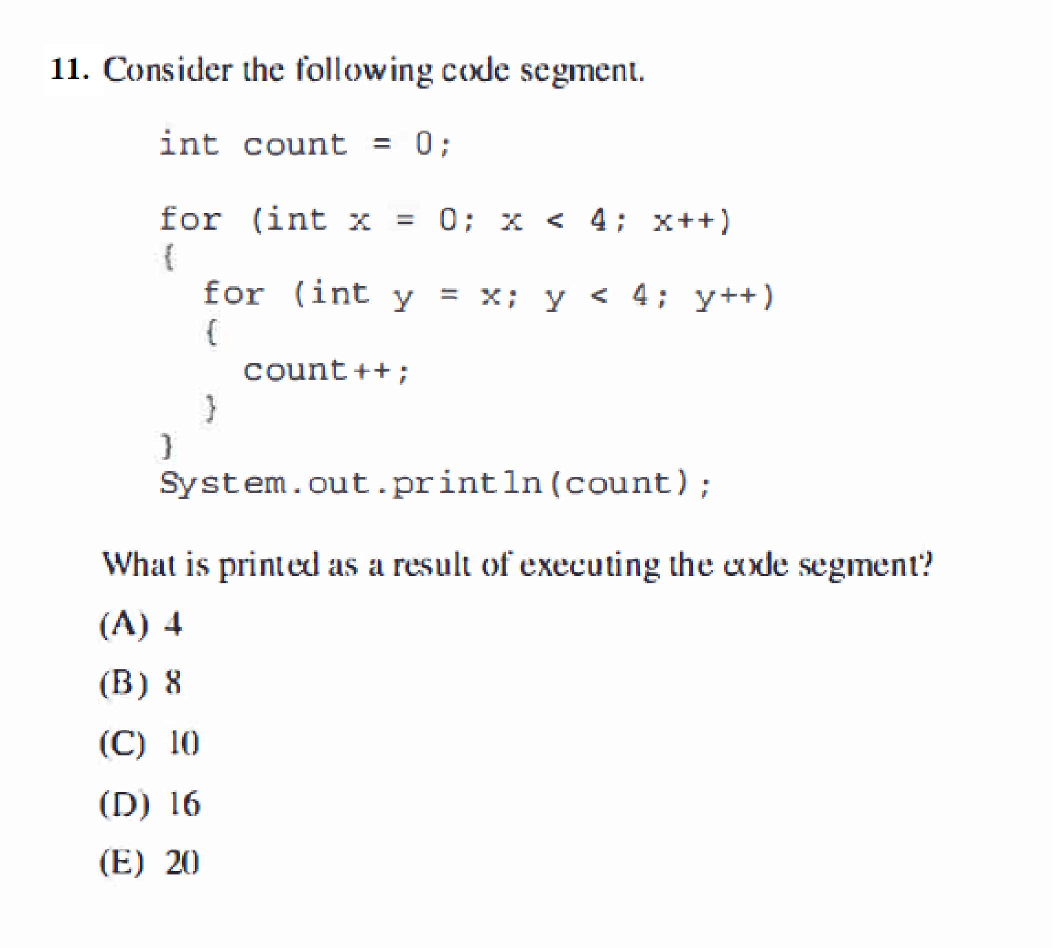
(A) `Value is: 21`

(B) `Value is: 2.3333333`

(C) `Value is: 2`

(D) `Value is: 0`

(E) `Value is: 1`



11. Consider the following code segment.

```
int count = 0;

for (int x = 0; x < 4; x++)
{
  for (int y = x; y < 4; y++)
  {
    count++;
  }
}
System.out.println(count);
```

What is printed as a result of executing the code segment?

(A) 4

(B) 8

(C) 10

(D) 16

(E) 20

12. Assume that `a` and `b` have been defined and initialized as `int` values. The expression

```
!(!(a != b ) && (b > 7))
```

is equivalent to which of the following?

(A) `(a != b) || (b < 7)`

(B) `(a != b) || (b <= 7)`

(C) `(a == b) || (b <= 7)`

(D) `(a != b) && (b <= 7)`

(E) `(a == b) && (b > 7)`

13. Consider the following code segment.

```
for (int k = 1; k <= 100; k++)
  if ((k % 4) == 0)
    System.out.println(k);
```

Which of the following code segments will produce the same output as the code segment above?

(A)
```
for (int k = 1; k <= 25; k++)
  System.out.println(k);
```

(B)
```
for (int k = 1; k <= 100; k = k + 4)
  System.out.println(k);
```

(C)
```
for (int k = 1; k <= 100; k++)
  System.out.println(k % 4);
```

(D)
```
for (int k = 4; k <= 25; k = 4 * k)
  System.out.println(k);
```

(E)
```
for (int k = 4; k <= 100; k = k + 4)
  System.out.println(k);
```

14. Consider the following code segment.

```
int sum = 0;
int k = 1;
while (sum < 12 || k < 4)
  sum += k;

System.out.println(sum);
```

What is printed as a result of executing the code segment?

(A) 6

(B) 10

(C) 12

(D) 15

(E) Nothing is printed due to an infinite loop.

15. Consider the following code segment.

```
int num = 2574;
int result = 0;

while (num > 0)
{
  result = result * 10 + num % 10;
  num /= 10;
}
System.out.println(result);
```

What is printed as a result of executing the code segment?

(A) 2

(B) 4

(C) 18

(D) 2574

(E) 4752

16. Consider the following code segment.

```
int x = 1;
while ( /* missing code */ )
{
  System.out.print(x + " ");
  x = x + 2;
}
```

Consider the following possible replacements for `/* missing code */`.

I. `x < 6`

II. `x != 6`

III. `x < 7`

Which of the proposed replacements for `/* missing code */` will cause the code segment to print only the values `1 3 5` ?

(A) I only

(B) II only

(C) I and II only

(D) I and III only

(E) I, II, and III

17. Assume that `x` and `y` have been declared and initialized with `int` values. Consider the following Java expression.

```
(y > 10000) || (x > 1000 && x < 1500)
```

Which of the following is equivalent to the expression given above?

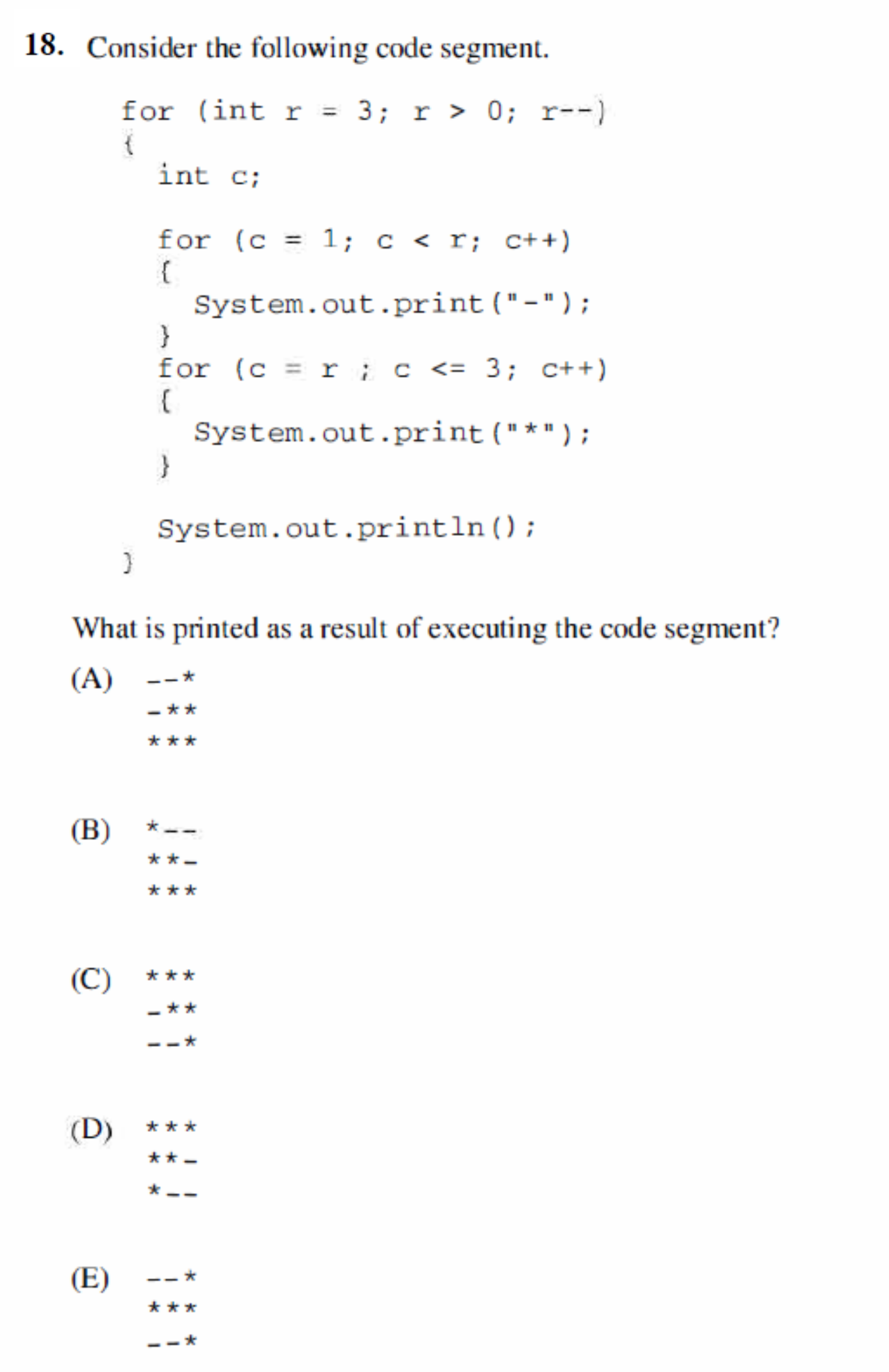
(A) `(y > 10000 || x > 1000) && (y > 10000 || x < 1500)`

(B) `(y > 10000 || x > 1000) || (y > 10000 || x < 1500)`

(C) `(y > 10000) && (x > 1000 || x < 1500)`

(D) `(y > 10000 && x > 1000) || (y > 10000 && x < 1500)`

(E) `(y > 10000 && x > 1000) && (y > 10000 && x < 1500)`



18. Consider the following code segment.

```
for (int r = 3; r > 0; r--)
{
  int c;

  for (c = 1; c < r; c++)
  {
    System.out.print("-");
  }
  for (c = r ; c <= 3; c++)
  {
    System.out.print("*");
  }

  System.out.println();
}
```

What is printed as a result of executing the code segment?

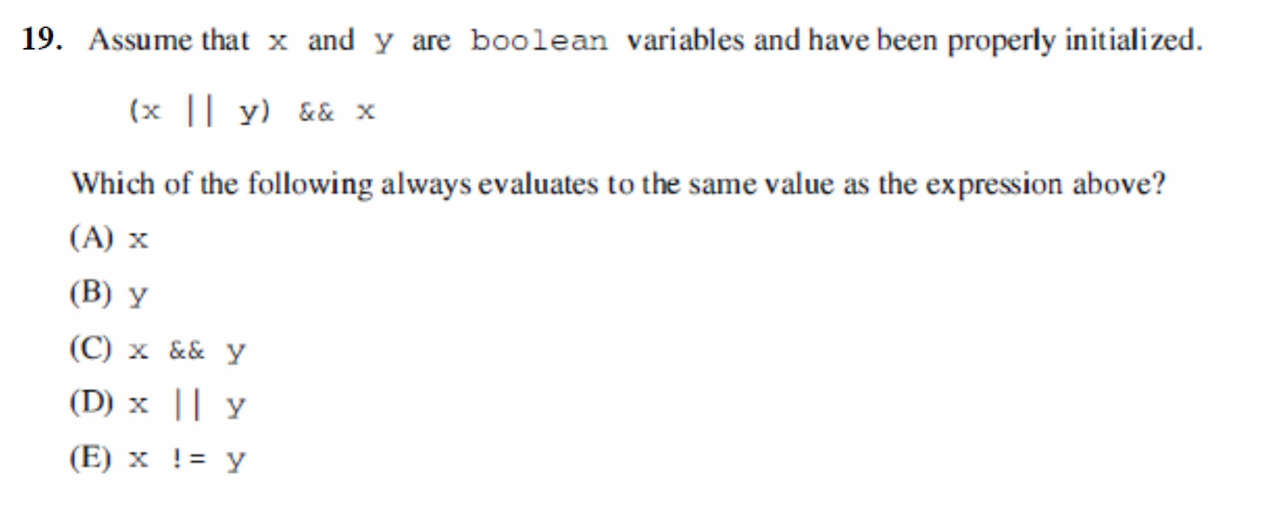
(A)
```
--*
-**
***
```

(B)
```
*--
**-
***
```

(C)
```
***
-**
--*
```

(D)
```
***
**-
*--
```

(E)
```
--*
***
--*
```



19. Assume that `x` and `y` are `boolean` variables and have been properly initialized.

```
(x || y) && x
```

Which of the following always evaluates to the same value as the expression above?

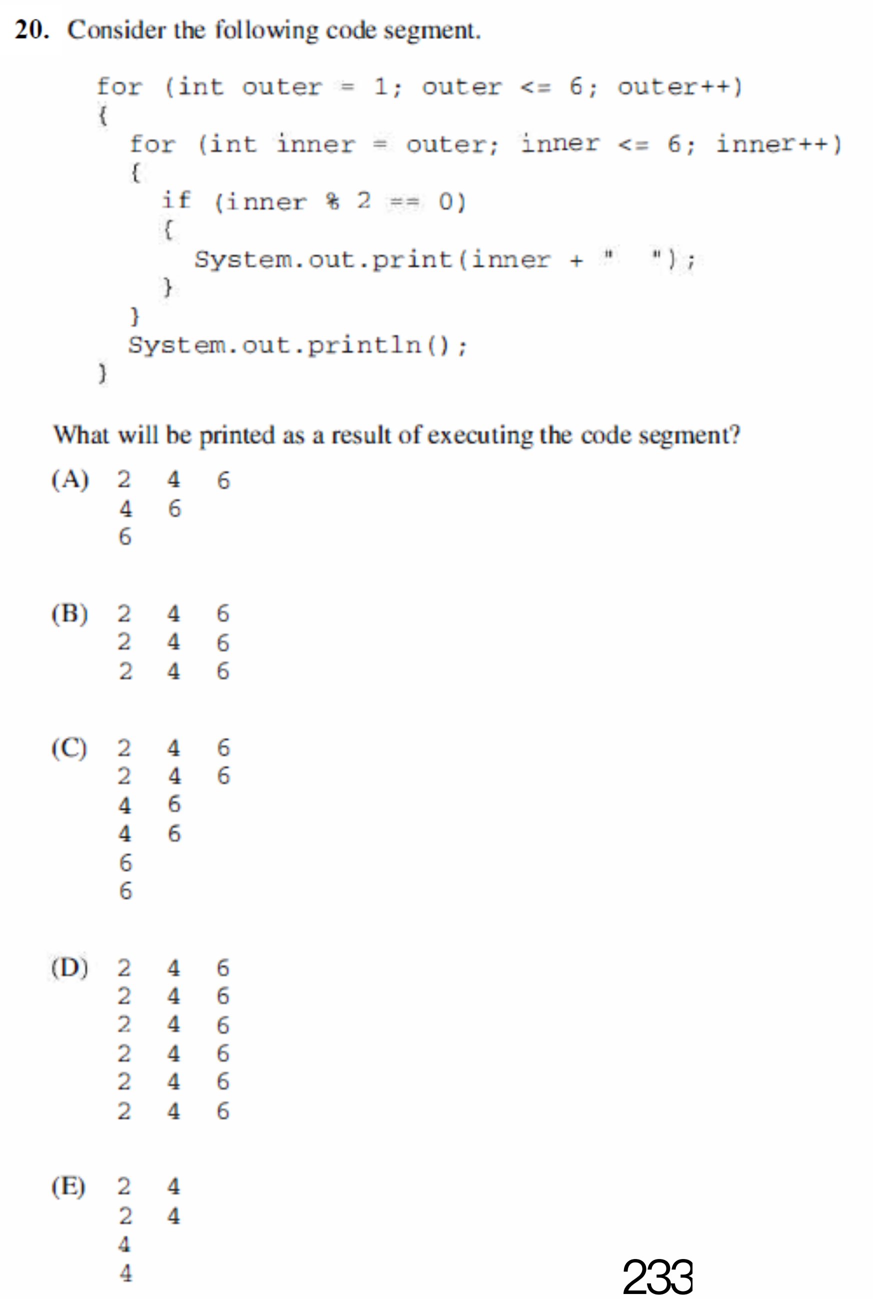
(A) `x`

(B) `y`

(C) `x && y`

(D) `x || y`

(E) `x != y`



20. Consider the following code segment.

```
for (int outer = 1; outer <= 6; outer++)
{
  for (int inner = outer; inner <= 6; inner++)
  {
    if (inner % 2 == 0)
    {
      System.out.print(inner + "  ");
    }
  }
  System.out.println();
}
```

What will be printed as a result of executing the code segment?

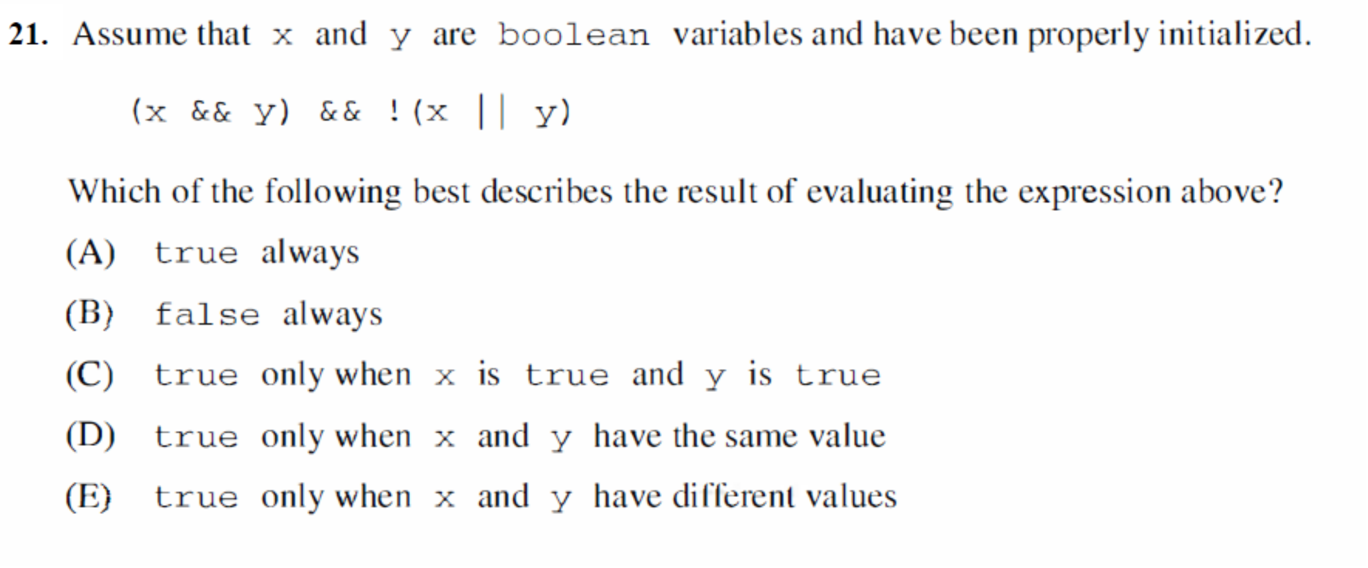
(A)
```
2   4   6
4   6
6
```

(B)
```
2   4   6
2   4   6
2   4   6
```

(C)
```
2   4   6
2   4   6
4   6
4   6
6
6
```

(D)
```
2   4   6
2   4   6
2   4   6
2   4   6
2   4   6
2   4   6
```

(E)
```
2   4
2   4
4
4
```



21. Assume that `x` and `y` are `boolean` variables and have been properly initialized.

```
(x && y) && !(x || y)
```

Which of the following best describes the result of evaluating the expression above?

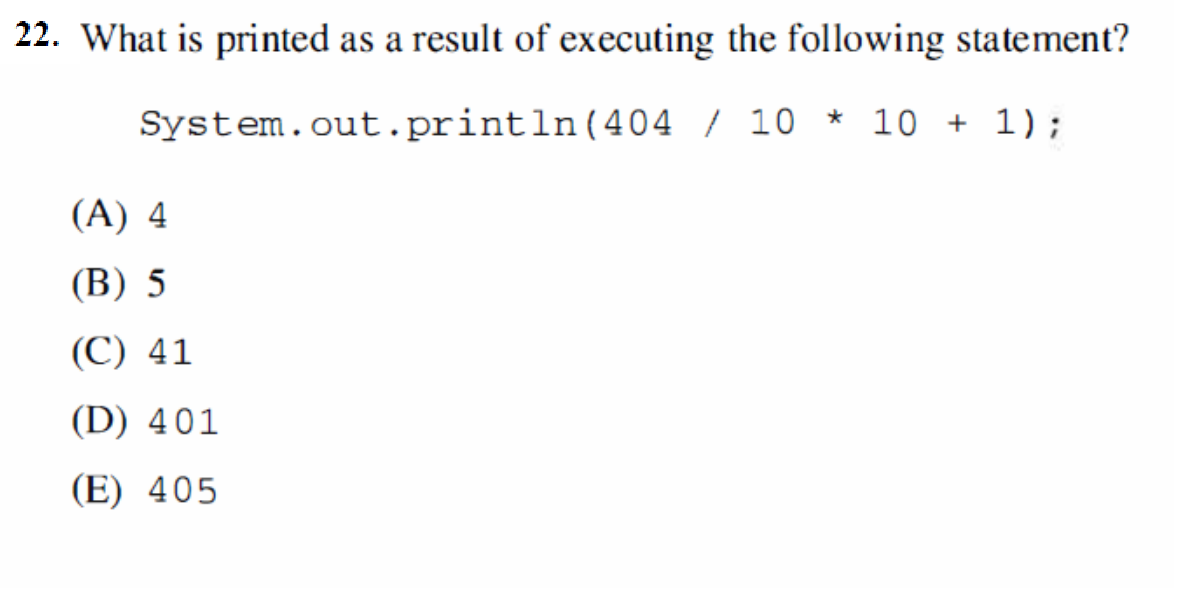
(A) `true` always

(B) `false` always

(C) `true` only when `x` is `true` and `y` is `true`

(D) `true` only when `x` and `y` have the same value

(E) `true` only when `x` and `y` have different values



22. What is printed as a result of executing the following statement?

```
System.out.println(404 / 10 * 10 + 1);
```

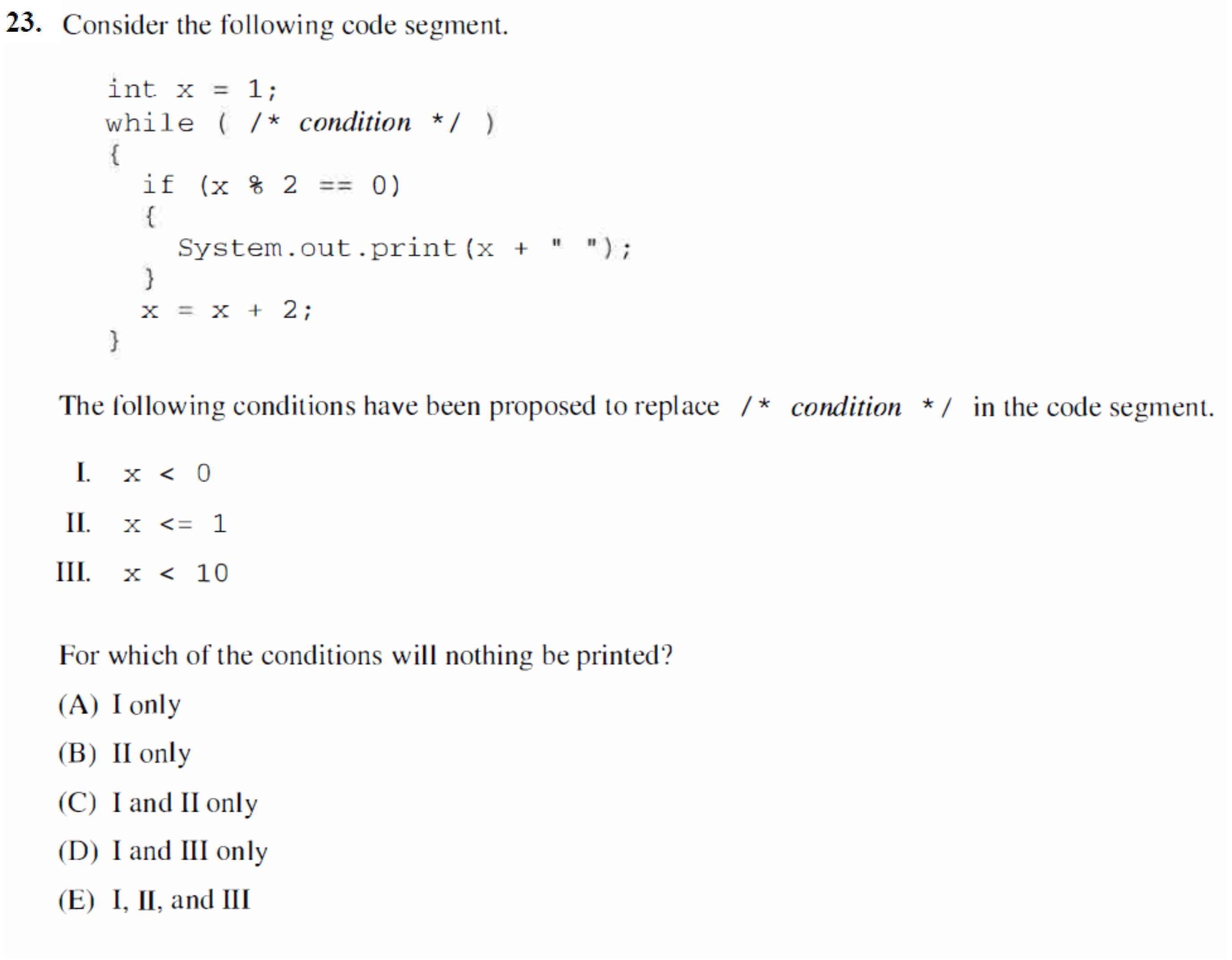
(A) 4

(B) 5

(C) 41

(D) 401

(E) 405



23. Consider the following code segment.

```
int x = 1;
while ( /* condition */ )
{
  if (x % 2 == 0)
  {
    System.out.print(x + " ");
  }
  x = x + 2;
}
```

The following conditions have been proposed to replace `/* condition */` in the code segment.

I. `x < 0`

II. `x <= 1`

III. `x < 10`

For which of the conditions will nothing be printed?

(A) I only

(B) II only

(C) I and II only

(D) I and III only

(E) I, II, and III

Basic Part-2 Multiple Choice

1. Consider the following method.

```
public static void message(int a, int b, int c)
{
   if (a < 10)
   {
      if (b < 10)
      {
         System.out.print("X");
      }
      System.out.print("Y");
   }
   if (c < 10)
   {
      if (b > 10)
      {
         System.out.print("Y");
      }
      else
      {
         System.out.print("Z");
      }
   }
}
```

What is printed as a result of the call `message(5, 15, 5)` ?

(A) `XY`

(B) `XYZ`

(C) `Y`

(D) `YY`

(E) `Z`

2. Consider the following method.

```
public static void printSome(int num1, int num2)
{
   for (int i = 0; i < num1; i++)
   {
      if (i % num2 == 0 && i % 2 == 0)
      {
         System.out.print(i + " ");
      }
   }
}
```

Which of the following method calls will cause `"0 10 "` to be printed?

(A) `printSome(0, 20)`

(B) `printSome(5, 10)`

(C) `printSome(10, 5)`

(D) `printSome(20, 5)`

(E) `printSome(25, 5)`

3. The following categories are used by some researchers to categorize zip codes as urban, suburban, or rural based on population density.

- An urban zip code is a zip code with more than 3,000 people per square mile.
- A suburban zip code is a zip code with between 1,000 and 3,000 people, inclusive, per square mile.
- A rural zip code is a zip code with fewer than 1,000 people per square mile.

Consider the following method, which is intended to categorize a zip code as urban, suburban, or rural based on the population density of the area included in the zip code.

```
public static String getCategory(int density)
{
   /* missing code */
}
```

Which of the following code segments can replace `/* missing code */` so the `getCategory` method works as intended?

```
 I.   String cat;
      if (density > 3000)
      {
         cat = "urban";
      }
      else if (density > 999)
      {
         cat = "suburban";
      }
      else
      {
         cat = "rural";
      }
      return cat;
```

```
II.   String cat;
      if (density > 3000)
      {
         cat = "urban";
      }
      if (density > 999)
      {
         cat = "suburban";
      }
      cat = "rural";
      return cat;
```

III.
```
if (density > 3000)
{
   return "urban";
}
if (density > 999)
{
   return "suburban";
}
return "rural";
```

(A) I only

(B) III only

(C) I and II only

(D) I and III only

(E) I, II, and III

4. Consider the following methods.

```
/** Precondition: a > 0 and b > 0 */
public static int methodOne(int a, int b)
{
   int loopCount = 0;
   for (int i = 0; i < a / b; i++)
   {
      loopCount++;
   }
   return loopCount;
}

/** Precondition: a > 0 and b > 0 */
public static int methodTwo(int a, int b)
{
   int loopCount = 0;
   int i = 0;
   while (i < a)
   {
      loopCount++;
      i += b;
   }
   return loopCount;
}
```

Which of the following best describes the conditions under which `methodOne` and `methodTwo` return the same value?

(A) When `a` and `b` are both even

(B) When `a` and `b` are both odd

(C) When `a` is even and `b` is odd

(D) When `a % b` is equal to zero

(E) When `a % b` is equal to one

Questions 5-6 refer to the following method.

```
public static int mystery(int n)
{
  int x = 1;
  int y = 1;

  // Point A

  while (n > 2)
  {
    x = x + y;

    // Point B

    y = x - y;
    n--;
  }

  // Point C

  return x;
}
```

5. What value is returned as a result of the call `mystery(6)` ?

(A) 1

(B) 5

(C) 6

(D) 8

(E) 13

---

6. Which of the following is true of method `mystery` ?

(A) `x` will sometimes be `1` at `// Point B`.

(B) `x` will never be `1` at `// Point C`.

(C) `n` will never be greater than `2` at `// Point A`.

(D) `n` will sometimes be greater than `2` at `// Point C`.

(E) `n` will always be greater than `2` at `// Point B`.

7. Consider the following method.

```
public int compute(int n, int k)
{
  int answer = 1;

  for (int i = 1; i <= k; i++)
    answer *= n;

  return answer;
}
```

Which of the following represents the value returned as a result of the call `compute(n, k)` ?

(A) `n*k`

(B) `n!`

(C) n^k

(D) 2^k

(E) k^n

8. Consider the following method.

```
public void test(int x)
{
  int y;

  if (x % 2 == 0)
    y = 3;
  else if (x > 9)
    y = 5;
  else
    y = 1;

  System.out.println("y = " + y);
}
```

Which of the following test data sets would test each possible output for the method?

(A) 8, 9, 12

(B) 7, 9, 11

(C) 8, 9, 11

(D) 8, 11, 13

(E) 7, 9, 10

9. Consider the following incomplete method, which is intended to return the number of integers that evenly divide the integer `inputVal`. Assume that `inputVal` is greater than 0.

```
public static int numDivisors(int inputVal)
{
  int count = 0;
  for (int k = 1; k <= inputVal; k++)
  {
    if ( /* condition */ )

      count++;


  return count;
```

Which of the following can be used to replace `/* condition */` so that `numDivisors` will work as intended?

(A) `inputVal % k == 0`

(B) `k % inputVal == 0`

(C) `inputVal % k != 0`

(D) `inputVal / k == 0`

(E) `k / inputVal > 0`

10. Consider the following method, `biggest`, which is intended to return the greatest of three integers. It does not always work as intended.

```
public static int biggest(int a, int b, int c)
{
  if ((a > b) && (a > c))
  {
    return a;
  }
  else if ((b > a) && (b > c))
  {
    return b;
  }
  else
  {
    return c;
  }
}
```

Which of the following best describes the error in the method?

(A) `biggest` always returns the value of `a`.

(B) `biggest` may not work correctly when `c` has the greatest value.

(C) `biggest` may not work correctly when `a` and `b` have equal values.

(D) `biggest` may not work correctly when `a` and `c` have equal values.

(E) `biggest` may not work correctly when `b` and `c` have equal values.

11. Consider the following method.

```
//* Precondition: num > 0 */
public static int doWhat(int num)
{
  int var = 0;

  for (int loop = 1; loop <= num; loop = loop + 2)
  {
    var += loop;
  }

  return var;
}
```

Which of the following best describes the value returned from a call to `doWhat` ?

(A) `num`

(B) The sum of all integers between `1` and `num`, inclusive

(C) The sum of all even integers between `1` and `num`, inclusive

(D) The sum of all odd integers between `1` and `num`, inclusive

(E) No value is returned because of an infinite loop.

12. The price per box of ink pens advertised in an office supply catalog is based on the number of boxes ordered. The following table shows the pricing.

Number of Boxes	Price per Box
1 up to but not including 5	$5.00
5 up to but not including 10	$3.00
10 or more	$1.50

The following incomplete method is intended to return the total cost of an order based on the value of the parameter `numBoxes`.

```
/** Precondition: numBoxes > 0 */
public static double getCost(int numBoxes)
{
  double totalCost = 0.0;

  /* missing code */

  return totalCost;
}
```

Which of the following code segments can be used to replace `/*` *missing code* `*/` so that method `getCost` will work as intended?

```
 I.  if (numBoxes >= 10)
     {
       totalCost = numBoxes * 1.50;
     }
     if (numBoxes >= 5)
     {
       totalCost = numBoxes * 3.00;
     }
     if (numBoxes > 0)
     {
       totalCost = numBoxes * 5.00;
     }
```

```
 II. if (numBoxes >= 10)
     {
       totalCost = numBoxes * 1.50;
     }
     else if (numBoxes >= 5)
     {
       totalCost = numBoxes * 3.00;
     }
     else
     {
       totalCost = numBoxes * 5.00;
     }
```

```
III. if (numBoxes > 0)
     {
       totalCost = numBoxes * 5.00;
     }
     else if (numBoxes >= 5)
     {
       totalCost = numBoxes * 3.00;
     }
     else if (numBoxes >= 10)
     {
       totalCost = numBoxes * 1.50;
     }
```

(A) I only

(B) II only

(C) III only

(D) I and II

(E) II and III

1D Array Multiple Choice

1. Consider the following method, which is intended to return the number of *local maximum* values in an array. Local maximum values are array elements that are greater than both adjacent array elements. The first and last elements of an array have only a single adjacent element, so neither the first nor the last array element is counted by this method. For example, an array containing the values `{3, 9, 7, 4, 10, 12, 3, 8}` has two local maximum values: `9` and `12`.

```
public static int countPeaks(int[] data)
{
   int numPeaks = 0;

   for ( /* missing loop header */ )
   {
      if (data[p - 1] < data[p] && data[p] > data[p + 1])
      {
         numPeaks++;
      }
   }
   return numPeaks;
}
```

Which of the following can replace `/*` *missing loop header* `*/` so the method `countPeaks` works as intended?

(A) `int p = data.length - 1; p > 0; p--`

(B) `int p = 0; p < data.length; p++`

(C) `int p = 0; p < data.length - 1; p++`

(D) `int p = 1; p < data.length; p++`

(E) `int p = 1; p < data.length - 1; p++`

2. Consider an integer array `nums`, which has been properly declared and initialized with one or more values. Which of the following code segments counts the number of negative values found in `nums` and stores the count in `counter` ?

I.
```
int counter = 0;
int i = -1;
while (i <= nums.length - 2)
{
   i++;
   if (nums[i] < 0)
   {
      counter++;
   }
}
```

II.
```
int counter = 0;
for (int i = 1; i < nums.length; i++)
{
   if (nums[i] < 0)
   {
      counter++;
   }
}
```

III.
```
int counter = 0;
for (int i : nums)
{
   if (nums[i] < 0)
   {
      counter++;
   }
}
```

(A) I only

(B) II only

(C) I and II only

(D) I and III only

(E) I, II, and III

3. Consider the `mode` method, which is intended to return the most frequently occurring value (mode) in its `int[]` parameter `arr`. For example, if the parameter of the `mode` method has the contents `{6, 5, 1, 5, 2, 6, 5}`, then the method is intended to return `5`.

```
/** Precondition: arr.length >= 1 */
public static int mode(int[] arr)
{
   int modeCount = 1;
   int mode = arr[0];

   for (int j = 0; j < arr.length; j++)
   {
      int valCount = 0;
      for (int k = 0; k < arr.length; k++)
      {
         if ( /* missing condition 1 */ )
         {
            valCount++;
         }
      }
      if ( /* missing condition 2 */ )
      {
         modeCount = valCount;
         mode = arr[j];
      }
   }
   return mode;
}
```

Which of the following can replace `/*` *missing condition 1* `*/` and `/*` *missing condition 2* `*/` so the code segment works as intended?

	/* *missing condition 1* */	/* *missing condition 2* */
(A)	`arr[j] == arr[k]`	`valCount > modeCount`
(B)	`arr[j] == arr[k]`	`modeCount > valCount`
(C)	`arr[j] != arr[k]`	`valCount > modeCount`
(D)	`arr[j] != arr[k]`	`modeCount > valCount`
(E)	`arr[j] != arr[k]`	`modeCount != valCount`

4. Consider the following method.

```
public static int mystery(int[] arr)
{
  int x = 0;

  for (int k = 0; k < arr.length; k = k + 2)
    x = x + arr[k];

  return x;
}
```

Assume that the array `nums` has been declared and initialized as follows.

```
int[] nums = {3, 6, 1, 0, 1, 4, 2};
```

What value will be returned as a result of the call `mystery(nums)` ?

(A) 5

(B) 6

(C) 7

(D) 10

(E) 17

5. Consider the following method that is intended to return the sum of the elements in the array `key`.

```
public static int sumArray(int[] key)
{
  int sum = 0;

  for (int i = 1; i <= key.length; i++)
  {
    /* missing code */
  }

  return sum;
}
```

Which of the following statements should be used to replace `/* missing code */` so that `sumArray` will work as intended?

(A) `sum = key[i];`

(B) `sum += key[i - 1];`

(C) `sum += key[i];`

(D) `sum += sum + key[i - 1];`

(E) `sum += sum + key[i];`

6. Consider the following code segment.

```
int[] arr = {7, 2, 5, 3, 0, 10};
for (int k = 0; k < arr.length - 1; k++)
{
  if (arr[k] > arr[k + 1])
    System.out.print(k + "  " + arr[k] + "  ");
}
```

What will be printed as a result of executing the code segment?

(A) 0 2 2 3 3 0

(B) 0 7 2 5 3 3

(C) 0 7 2 5 5 10

(D) 1 7 3 5 4 3

(E) 7 2 5 3 3 0

7. Consider the following incomplete method that is intended to return a string formed by concatenating elements from the parameter `words`. The elements to be concatenated start with `startIndex` and continue through the last element of `words` and should appear in reverse order in the resulting string.

```
/** Precondition: words.length > 0;
 *                startIndex >= 0
 */
public static String concatWords(String[] words, int startIndex)
{
  String result = "";

  /* missing code */

  return result;
}
```

For example, the following code segment uses a call to the `concatWords` method.

```
String[] things = {"Bear", "Apple", "Gorilla", "House", "Car"};
System.out.println(concatWords(things, 2));
```

When the code segment is executed, the string `"CarHouseGorilla"` is printed.

The following three code segments have been proposed as replacements for `/*` *missing code* `*/`.

```
I.     for (int k = startIndex; k < words.length; k++)
       {
         result += words[k] + words[words.length - k - 1];
       }

II.    int k = words.length - 1;
       while (k >= startIndex)
       {
         result += words[k];
         k--;
       }

III.   String[] temp = new String[words.length];
       for (int k = 0; k <= words.length / 2; k++)
       {
         temp[k] = words[words.length - k - 1];
         temp[words.length - k - 1] = words[k];
       }

       for (int k = 0; k < temp.length - startIndex; k++)
       {
         result += temp[k];
       }
```

Which of these code segments can be used to replace `/*` *missing code* `*/` so that `concatWords` will work as intended?

(A) I only

(B) II only

(C) III only

(D) I and II

(E) II and III

8. Consider the following method, `isSorted`, which is intended to return `true` if an array of integers is sorted in nondecreasing order and to return `false` otherwise.

```
/** @param data  an array of integers
 *  @return true  if the values in the array appear in sorted (nondecreasing) order
 */
public static boolean isSorted(int[] data)
{
  /* missing code */
}
```

Which of the following can be used to replace `/* missing code */` so that `isSorted` will work as intended?

I.
```
for (int k = 1; k < data.length; k++)
{
  if (data[k - 1] > data[k])
    return false;
}
return true;
```

II.
```
for (int k = 0; k < data.length; k++)
{
  if (data[k] > data[k + 1])
    return false;
}
return true;
```

III.
```
for (int k = 0; k < data.length - 1; k++)
{
  if (data[k] > data[k + 1])
    return false;
  else
    return true;
}
return true;
```

(A) I only

(B) II only

(C) III only

(D) I and II only

(E) I and III only

9. Consider the following incomplete method that is intended to return an array that contains the contents of its first array parameter followed by the contents of its second array parameter.

```
public static int[] append(int[] a1, int[] a2)
{
  int[] result = new int[a1.length + a2.length];

  for (int j = 0; j < a1.length; j++)
    result[j] = a1[j];

  for (int k = 0; k < a2.length; k++)
    result[ /* index */ ] = a2[k];

  return result;
}
```

Which of the following expressions can be used to replace `/* index */` so that `append` will work as intended?

(A) `j`

(B) `k`

(C) `k + a1.length - 1`

(D) `k + a1.length`

(E) `k + a1.length + 1`

10. Consider the following code segment.

```
int[] arr = {1, 2, 3, 4, 5, 6, 7};

for (int k = 3; k < arr.length - 1; k++)
  arr[k] = arr[k + 1];
```

Which of the following represents the contents of `arr` as a result of executing the code segment?

(A) `{1, 2, 3, 4, 5, 6, 7}`

(B) `{1, 2, 3, 5, 6, 7}`

(C) `{1, 2, 3, 5, 6, 7, 7}`

(D) `{1, 2, 3, 5, 6, 7, 8}`

(E) `{2, 3, 4, 5, 6, 7, 7}`

11. Consider the following method.

```
public static void arrayMethod(int nums[])
{
  int j = 0;
  int k = nums.length - 1;

  while (j < k)
  {
    int x = nums[j];
    nums[j] = nums[k];
    nums[k] = x;
    j++;
    k--;
  }
}
```

Which of the following describes what the method `arrayMethod()` does to the array `nums`?

(A) The array `nums` is unchanged.

(B) The first value in `nums` is copied to every location in the array.

(C) The last value in `nums` is copied to every location in the array.

(D) The method generates an `ArrayIndexOutOfBoundsException`.

(E) The contents of the array `nums` are reversed.

12. Assume that the array `arr` has been defined and initialized as follows.

```
int[] arr = /* initial values for the array */ ;
```

Which of the following will correctly print all of the odd integers contained in `arr` but none of the even integers contained in `arr` ?

(A)
```
for (int x : arr)
  if (x % 2 != 0)
    System.out.println(x);
```

(B)
```
for (int k = 1; k < arr.length; k++)
  if (arr[k] % 2 != 0)
    System.out.println(arr[k]);
```

(C)
```
for (int x : arr)
  if (x % 2 != 0)
    System.out.println(arr[x]);
```

(D)
```
for (int k = 0; k < arr.length; k++)
  if (arr[k] % 2 != 0)
    System.out.println(k);
```

(E)
```
for (int x : arr)
  if (arr[x] % 2 != 0)
    System.out.println(arr[x]);
```

13. Consider the following method.

```
public void mystery(int[] data)
{
  for (int k = 0; k < data.length - 1; k++)
    data[k + 1] = data[k] + data[k + 1];
}
```

The following code segment appears in another method in the same class.

```
int[] values = {5, 2, 1, 3, 8};
mystery(values);
for (int v : values)
  System.out.print(v + " ");
System.out.println();
```

What is printed as a result of executing the code segment?

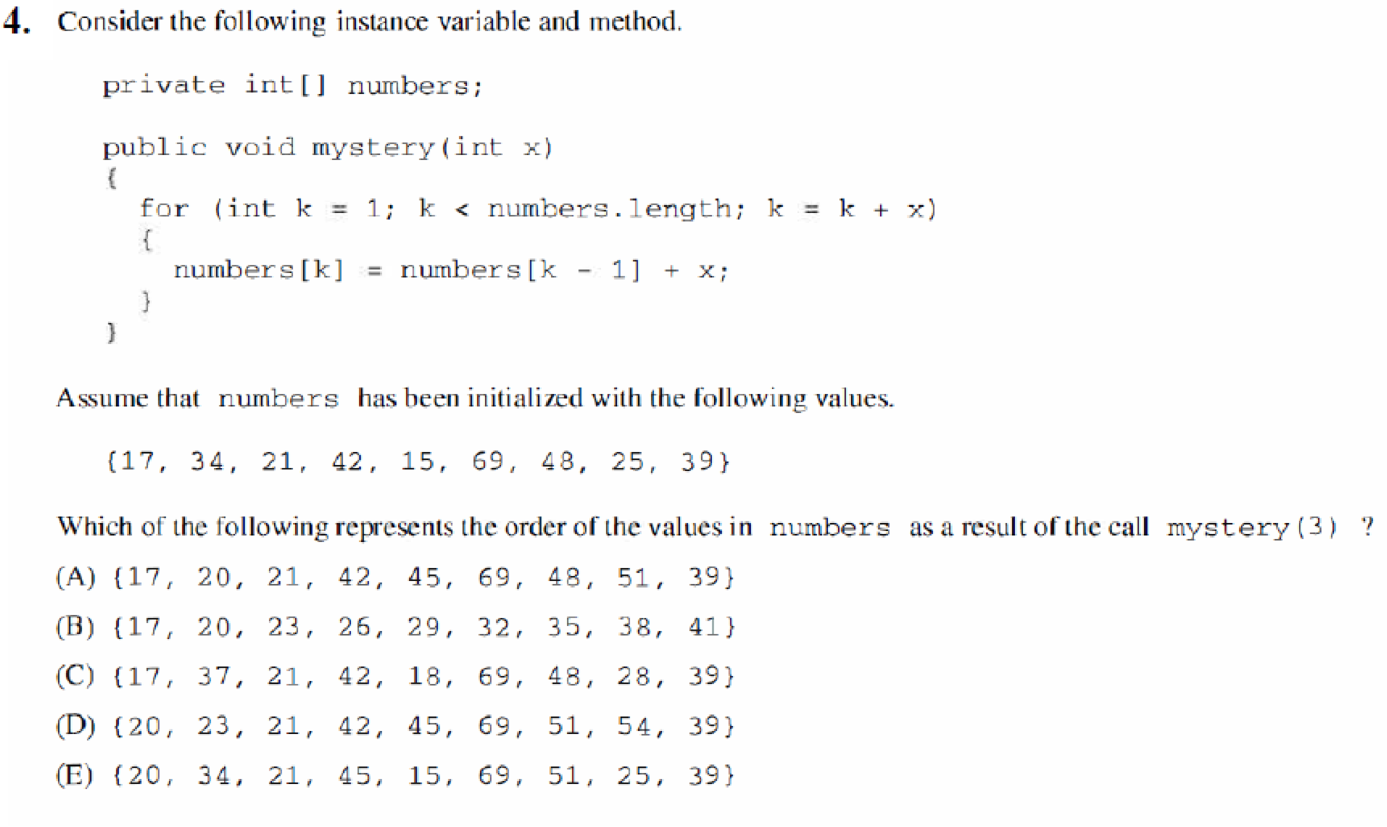
(A) `5 2 1 3 8`

(B) `5 7 3 4 11`

(C) `5 7 8 11 19`

(D) `7 3 4 11 8`

(E) Nothing is printed because an `ArrayIndexOutOfBoundsException` is thrown during the execution of method `mystery`.

14. 

Consider the following instance variable and method.

```
private int[] numbers;

public void mystery(int x)
{
  for (int k = 1; k < numbers.length; k = k + x)
  {
    numbers[k] = numbers[k - 1] + x;
  }
}
```

Assume that `numbers` has been initialized with the following values.

```
{17, 34, 21, 42, 15, 69, 48, 25, 39}
```

Which of the following represents the order of the values in `numbers` as a result of the call `mystery(3)` ?

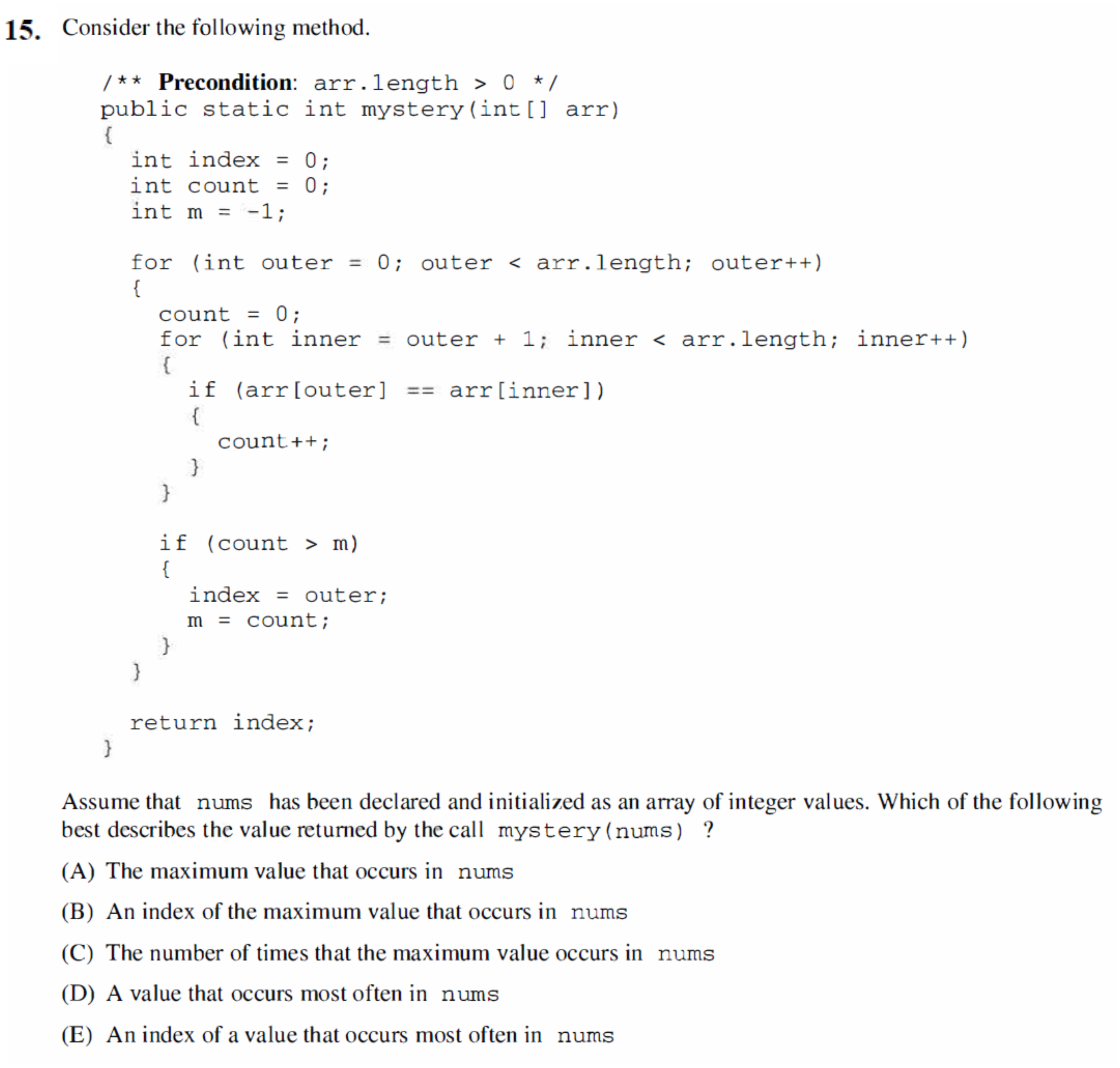
(A) `{17, 20, 21, 42, 45, 69, 48, 51, 39}`

(B) `{17, 20, 23, 26, 29, 32, 35, 38, 41}`

(C) `{17, 37, 21, 42, 18, 69, 48, 28, 39}`

(D) `{20, 23, 21, 42, 45, 69, 51, 54, 39}`

(E) `{20, 34, 21, 45, 15, 69, 51, 25, 39}`



15. Consider the following method.

```
/** Precondition: arr.length > 0 */
public static int mystery(int[] arr)
{
  int index = 0;
  int count = 0;
  int m = -1;

  for (int outer = 0; outer < arr.length; outer++)
  {
    count = 0;
    for (int inner = outer + 1; inner < arr.length; inner++)
    {
      if (arr[outer] == arr[inner])
      {
        count++;
      }
    }

    if (count > m)
    {
      index = outer;
      m = count;
    }
  }

  return index;
}
```

Assume that `nums` has been declared and initialized as an array of integer values. Which of the following best describes the value returned by the call `mystery(nums)` ?

(A) The maximum value that occurs in `nums`

(B) An index of the maximum value that occurs in `nums`

(C) The number of times that the maximum value occurs in `nums`

(D) A value that occurs most often in `nums`

(E) An index of a value that occurs most often in `nums`

2D-Array Multiple Choice

1. Consider the following code segment.

```
int[][] values = {{1, 2, 3}, {4, 5, 6}};
int x = 0;

for (int j = 0; j < values.length; j++)
{
   for (int k = 0; k < values[0].length; k++)
   {
      if (k == 0)
      {
         values[j][k] *= 2;
      }
      x += values[j][k];
   }
}
```

What is the value of `x` after the code segment is executed?

(A) `7`

(B) `17`

(C) `21`

(D) `26`

(E) `27`

2. Consider the following method, which is intended to print the values in its two-dimensional integer array parameter in row-major order.

```
public static void rowMajor(int[][] arr)
{
   /* missing code */
}
```

As an example, consider the following code segment.

```
int[][] theArray = {{1, 2}, {3, 4}, {5, 6}, {7, 8}};
rowMajor(theArray);
```

When executed, the code segment should produce the following output.

```
1 2 3 4 5 6 7 8
```

Which of the following code segments can replace /* *missing code* */ so that the rowMajor method works as intended?

(A)
```
for (int j : arr)
{
   for (int k : j)
   {
      System.out.print(j + " ");
   }
}
```

(B)
```
for (int j : arr)
{
   for (int k : j)
   {
      System.out.print(k + " ");
   }
}
```

(C)
```
for (int[] j : arr)
{
   for (int k : j)
   {
      System.out.print(j + " ");
   }
}
```

(D)
```
for (int[] j : arr)
{
   for (int k : j)
   {
      System.out.print(k + " ");
   }
}
```

(E)
```
for (int[] j : arr)
{
   for (int k : j)
   {
      System.out.print(arr[k] + " ");
   }
}
```

3. Consider the following code segment.

```
int[] oldArray = {1, 2, 3, 4, 5, 6, 7, 8, 9};
int[][] newArray = new int[3][3];

int row = 0;
int col = 0;
for (int value : oldArray)
{
  newArray[row][col] = value;
  row++;
  if ((row % 3) == 0)
  {
    col++;
    row = 0;
  }
}

System.out.println(newArray[0][2]);
```

What is printed as a result of executing the code segment?

(A) 3

(B) 4

(C) 5

(D) 7

(E) 8

4. Consider the following code segment.

```
String[][] letters = {{"A", "B", "C", "D"},
                      {"E", "F", "G", "H"},
                      {"I", "J", "K", "L"}};

for (int col = 1; col < letters[0].length; col++)
{
   for (int row = 1; row < letters.length; row++)
   {
      System.out.print(letters[row][col] + " ");
   }

   System.out.println();
}
```

What is printed as a result of executing this code segment?

(A)
```
A E I
F J
K
```

(B)
```
B F J
C G K
D H L
```

(C)
```
E I
F J
G K
H L
```

(D)
```
F G H
J K L
```

(E)
```
F J
G K
H L
```

5. Consider the following instance variable and method.

```
private int[] arr;

/** Precondition: arr.length > 0
 *  @return the largest value in array arr
 */
public int findMax()
{
  int maxVal = 0;

  for (int val : arr)
  {
    if (val > maxVal)
    {
      maxVal = val;
    }
  }

  return maxVal;
}
```

Method `findMax` is intended to return the largest value in the array `arr`. Which of the following best describes the conditions under which the method `findMax` will not work as intended?

(A) The largest value in `arr` occurs only once and is in `arr[0]`.

(B) The largest value in `arr` occurs only once and is in `arr[arr.length - 1]`.

(C) The largest value in `arr` is negative.

(D) The largest value in `arr` is zero.

(E) The largest value in `arr` occurs more than once.

6. Consider the following method.

```
public static int[] operation(int[][] matrix, int r, int c)
{
  int[] result = new int[matrix.length];

  for (int j = 0 ; j < matrix.length ; j++)
  {
    result[j] = matrix[r][j] * matrix[j][c];
  }
  return result;
}
```

The following code segment appears in another method in the same class.

```
int[][] mat = {{3, 2, 1, 4},
               {1, 2, 3, 4},
               {2, 2, 1, 2},
               {1, 1, 1, 1}};

int[] arr = operation(mat, 1, 2);
```

Which of the following represents the contents of `arr` as a result of executing the code segment?

(A) `{6, 4, 2, 4}`

(B) `{1, 6, 3, 4}`

(C) `{4, 3, 6, 1}`

(D) `{4, 4, 2, 2}`

(E) `{2, 2, 4, 4}`

7. Consider the following method.

```
/** Precondition: values has at least one row */
public static int calculate(int[][] values)
{
  int found = values[0][0];
  int result = 0;
  for (int[] row : values)
  {
    for (int y = 0; y < row.length; y++)
    {
      if (row[y] > found)
      {
        found = row[y];
        result = y;
      }
    }
  }
  return result;
}
```

Which of the following best describes what is returned by the `calculate` method?

(A) The largest value in the two-dimensional array

(B) The smallest value in the two-dimensional array

(C) The row index of an element with the largest value in the two-dimensional array

(D) The row index of an element with the smallest value in the two-dimensional array

(E) The column index of an element with the largest value in the two-dimensional array

8. Consider the following definition.

```
int[][] numbers = {{1, 2, 3},
                   {4, 5, 6}};
```

Which of the following code segments produces the output `123456` ?

(A)

```
for (int[] row : numbers)
{
  for (int n : row)
  {
    System.out.print(n);
  }
}
```

(B)

```
for (int[] row : numbers)
{
  for (int n : row)
  {
    System.out.print(row[n]);
  }
}
```

(C)

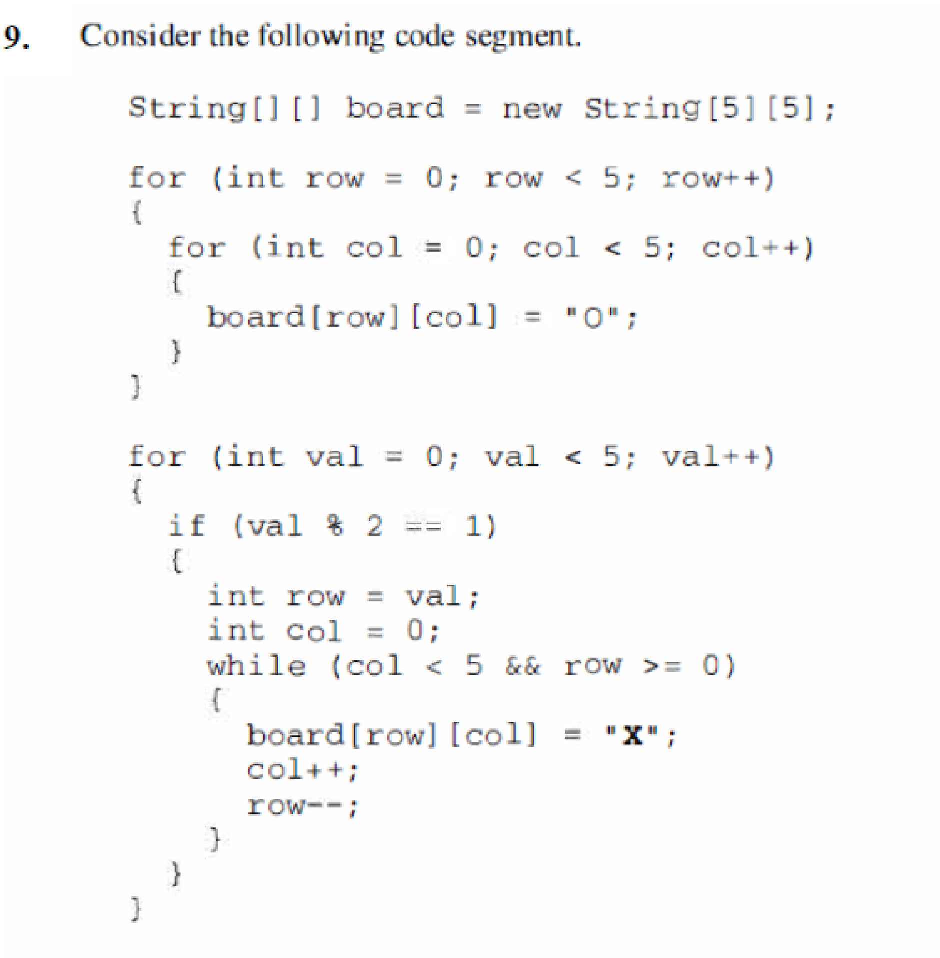
```
for (int rc = 0; rc < numbers.length; rc++)
{
  System.out.print(numbers[rc]);
}
```

(D)

```
for (int r = 0; r < numbers[0].length; r++)
{
  for (int c = 0; c < numbers.length; c++)
  {
    System.out.print(numbers[r][c]);
  }
}
```

(E)

```
for (int c = 0; c < numbers[0].length; c++)
{
  for (int r = 0; r < numbers.length; r++)
  {
    System.out.print(numbers[r][c]);
  }
}
```



9. Consider the following code segment.

```
String[][] board = new String[5][5];

for (int row = 0; row < 5; row++)
{
  for (int col = 0; col < 5; col++)
  {
    board[row][col] = "O";
  }
}

for (int val = 0; val < 5; val++)
{
  if (val % 2 == 1)
  {
    int row = val;
    int col = 0;
    while (col < 5 && row >= 0)
    {
      board[row][col] = "X";
      col++;
      row--;
    }
  }
}
```

Which of the following represents `board` after this code segment is executed?

(A)

	0	1	2	3	4
0	**X**	O	**X**	O	**X**
1	O	**X**	O	**X**	O
2	**X**	O	**X**	O	**X**
3	O	**X**	O	**X**	O
4	**X**	O	**X**	O	**X**

(B)

	0	1	2	3	4
0	O	**X**	O	**X**	O
1	**X**	O	**X**	O	**X**
2	O	**X**	O	**X**	O
3	**X**	O	**X**	O	**X**
4	O	**X**	O	**X**	O

(C)

	0	1	2	3	4
0	**X**	O	O	O	**X**
1	O	**X**	O	**X**	O
2	O	O	**X**	O	O
3	O	**X**	O	**X**	O
4	**X**	O	O	O	**X**

(D)

	0	1	2	3	4
0	O	**X**	O	O	O
1	O	O	**X**	O	O
2	**X**	O	O	**X**	O
3	O	**X**	O	O	**X**
4	O	O	**X**	O	O

(E)

	0	1	2	3	4
0	O	**X**	O	**X**	O
1	**X**	O	**X**	O	O
2	O	**X**	O	O	O
3	**X**	O	O	O	O
4	O	O	O	O	O

String Multiple-Choice

Questions 1 - 2 refer to the information below.

Consider the following method.

```
public static String[] strArrMethod(String[] arr)
{
   String[] result = new String[arr.length];

   for (int j = 0; j < arr.length; j++)
   {
      String sm = arr[j];
      for (int k = j + 1; k < arr.length; k++)
      {
         if (arr[k].length() < sm.length())
         {
            sm = arr[k];    // Line 12
         }
      }
      result[j] = sm;
   }
   return result;
}
```

1. Consider the following code segment.

```
String[] testOne = {"first", "day", "of", "spring"};
String[] resultOne = strArrMethod(testOne);
```

What are the contents of `resultOne` when the code segment has been executed?

(A) `{"day", "first", "of", "spring"}`

(B) `{"of", "day", "first", "spring"}`

(C) `{"of", "day", "of", "spring"}`

(D) `{"of", "of", "of", "spring"}`

(E) `{"spring", "first", "day", "of"}`

2. Consider the following code segment.

```
String[] testTwo = {"last", "day", "of", "the", "school", "year"};
String[] resultTwo = strArrMethod(testTwo);
```

How many times is the line labeled `//` Line 12 in the `strArrMethod` executed as a result of executing the code segment?

(A) 4 times

(B) 5 times

(C) 6 times

(D) 15 times

(E) 30 times

3. Consider the following method.

```
public static String rearrange(String str)
{
   String temp = "";

   for (int i = str.length() - 1; i > 0; i--)
   {
      temp += str.substring(i - 1, i);
   }

   return temp;
}
```

What, if anything, is returned by the method call `rearrange("apple")` ?

(A) `"appl"`

(B) `"apple"`

(C) `"elppa"`

(D) `"lppa"`

(E) Nothing is returned due to a run-time error.

4. Consider the following method.

```
public static String abMethod(String a, String b)
{
   int x = a.indexOf(b);

   while (x >= 0)
   {
      a = a.substring(0, x) + a.substring(x + b.length());
      x = a.indexOf(b);
   }

   return a;
}
```

What, if anything, is returned by the method call `abMethod("sing the song", "ng")` ?

(A) `"si"`

(B) `"si the so"`

(C) `"si the song"`

(D) `"sig the sog"`

(E) Nothing is returned because a `StringIndexOutOfBoundsException` is thrown.

5. Consider the following method.

```
public String mystery(String input)
{
  String output = "";

  for (int k = 1; k < input.length(); k = k + 2)
  {
    output += input.substring(k, k + 1);
  }

  return output;
}
```

What is returned as a result of the call `mystery("computer")` ?

(A) `"computer"`

(B) `"cmue"`

(C) `"optr"`

(D) `"ompute"`

(E) Nothing is returned because an `IndexOutOfBoundsException` is thrown.

6. Consider the following method.

```
public static String scramble(String word, int howFar)
{
  return word.substring(howFar + 1, word.length()) +
         word.substring(0, howFar);
}
```

What value is returned as a result of the call `scramble("compiler", 3)`?

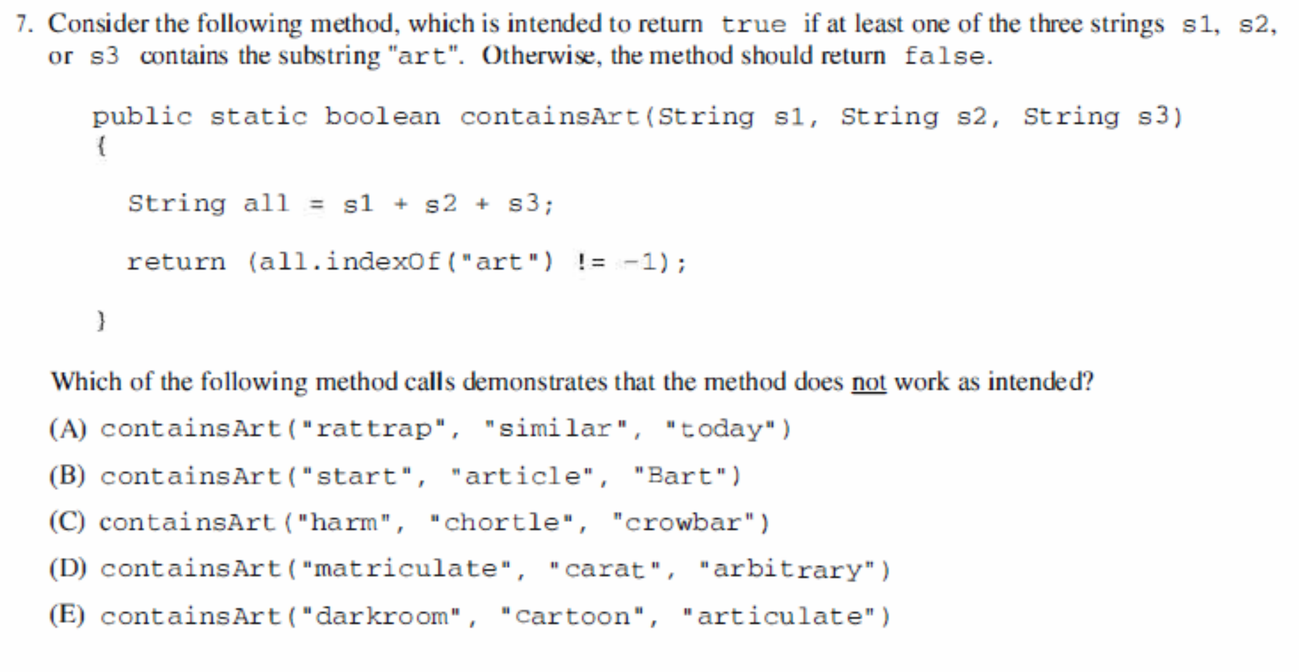
(A) `"compiler"`

(B) `"pilercom"`

(C) `"ilercom"`

(D) `"ilercomp"`

(E) No value is returned because an `IndexOutOfBoundsException` will be thrown.



7. Consider the following method, which is intended to return `true` if at least one of the three strings `s1`, `s2`, or `s3` contains the substring "`art`". Otherwise, the method should return `false`.

```
public static boolean containsArt(String s1, String s2, String s3)
{
  String all = s1 + s2 + s3;

  return (all.indexOf("art") != -1);

}
```

Which of the following method calls demonstrates that the method does not work as intended?

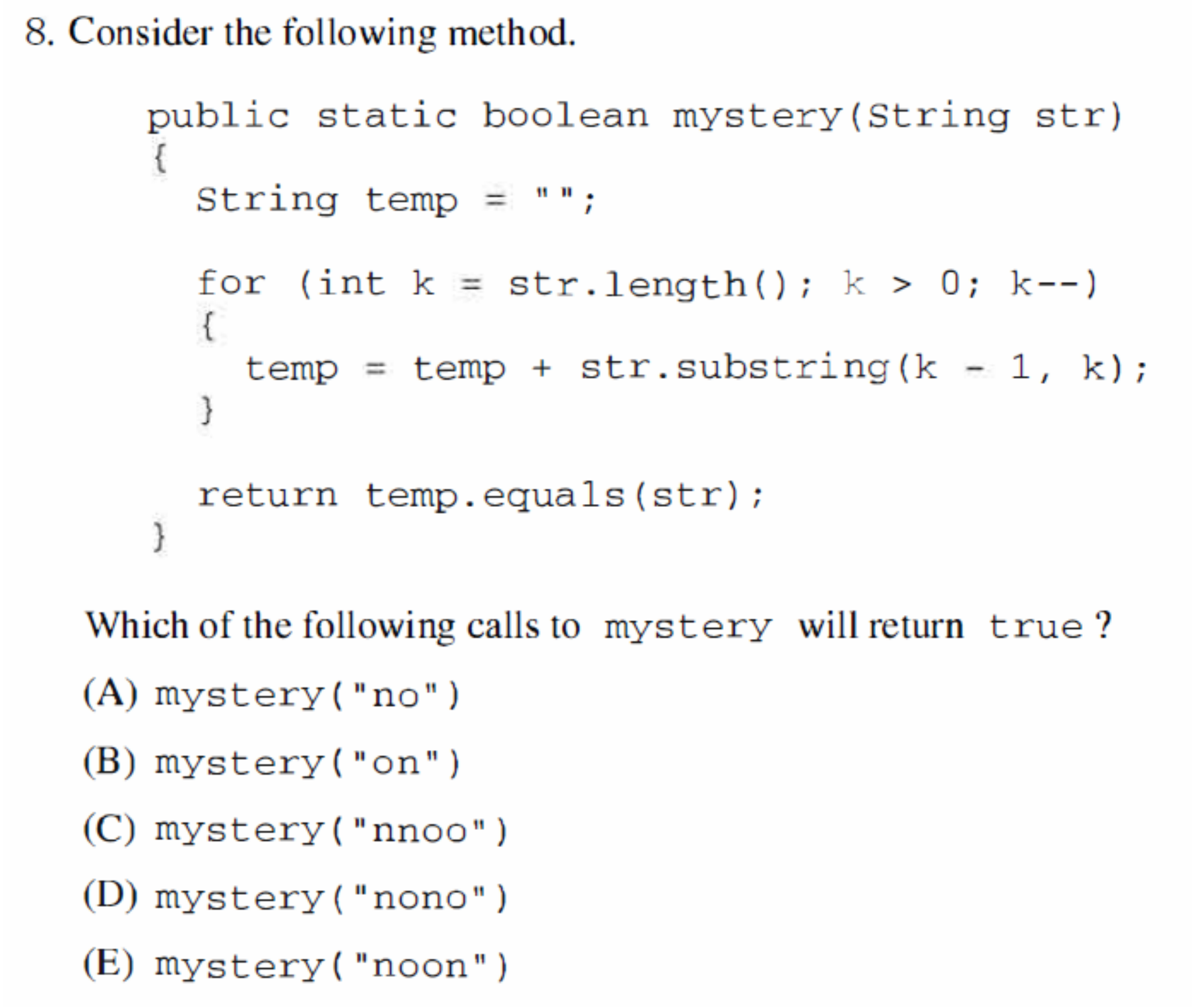
(A) `containsArt("rattrap", "similar", "today")`

(B) `containsArt("start", "article", "Bart")`

(C) `containsArt("harm", "chortle", "crowbar")`

(D) `containsArt("matriculate", "carat", "arbitrary")`

(E) `containsArt("darkroom", "cartoon", "articulate")`



8. Consider the following method.

```
public static boolean mystery(String str)
{
  String temp = "";

  for (int k = str.length(); k > 0; k--)
  {
    temp = temp + str.substring(k - 1, k);
  }

  return temp.equals(str);
}
```

Which of the following calls to `mystery` will return `true` ?

(A) `mystery("no")`

(B) `mystery("on")`

(C) `mystery("nnoo")`

(D) `mystery("nono")`

(E) `mystery("noon")`

Math Multiple-Choice

1. Consider the method `getHours`, which is intended to calculate the number of hours that a vehicle takes to travel between two *mile markers* on a highway if the vehicle travels at a constant speed of 60 miles per hour. A mile marker is a sign showing the number of miles along a road between some fixed location (for example, the beginning of a highway) and the current location.

   The following table shows two examples of the intended behavior of `getHours`, based on the `int` parameters `marker1` and `marker2`.

`marker1`	`marker2`	**Return Value**
`100`	`220`	`2.0`
`100`	`70`	`0.5`

   Consider the following implementation of `getHours`.

   ```
   public static double getHours(int marker1, int marker2)
   {
      /* missing statement */
      return hours;
   }
   ```

   Which of the following statements can replace `/*` *missing statement* `*/` so `getHours` works as intended?

   (A) `double hours = (Math.abs(marker1) - Math.abs(marker2)) / 60.0;`

   (B) `double hours = Math.abs(marker1 - marker2 / 60.0);`

   (C) `double hours = Math.abs(marker1 - marker2) / 60.0;`

   (D) `double hours = Math.abs((marker1 - marker2) / 60);`

   (E) `double hours = (double) (Math.abs(marker1 - marker2) / 60);`

2. Consider the following code segment. Assume that `a` is greater than zero.

```
int a = /* value not shown */;
int b = a + (int) (Math.random() * a);
```

Which of the following best describes the value assigned to `b` when the code segment is executed?

(A) `a`

(B) `2 * a`

(C) A random integer between `0` and `a - 1`, inclusive

(D) A random integer between `a` and `2 * a`, inclusive

(E) A random integer between `a` and `2 * a - 1`, inclusive

3. Consider the following method that is intended to determine if the `double` values `d1` and `d2` are close enough to be considered equal. For example, given a `tolerance` of `0.001`, the values `54.32271` and `54.32294` would be considered equal.

```
/** @return true if d1 and d2 are within the specified tolerance,
 *          false otherwise
 */
public boolean almostEqual(double d1, double d2, double tolerance)
{
  /* missing code */
}
```

Which of the following should replace `/* missing code */` so that `almostEqual` will work as intended?

(A) `return (d1 - d2) <= tolerance;`

(B) `return ((d1 + d2) / 2) <= tolerance;`

(C) `return (d1 - d2) >= tolerance;`

(D) `return ((d1 + d2) / 2) >= tolerance;`

(E) `return Math.abs(d1 - d2) <= tolerance;`

4. Consider the following method, which is intended to return the element of a 2-dimensional array that is closest in value to a specified number, `val`.

```
/** @return the element of 2-dimensional array mat whose value is closest to val */
public double findClosest(double[][] mat, double val)
{
  double answer = mat[0][0];
  double minDiff = Math.abs(answer - val);
  for (double[] row : mat)
  {
    for (double num : row)
    {
      if ( /* missing code */ )
      {
        answer = num;
        minDiff = Math.abs(num - val);
      }
    }
  }
  return answer;
}
```

Which of the following could be used to replace `/*` *missing code* `*/` so that `findClosest` will work as intended?

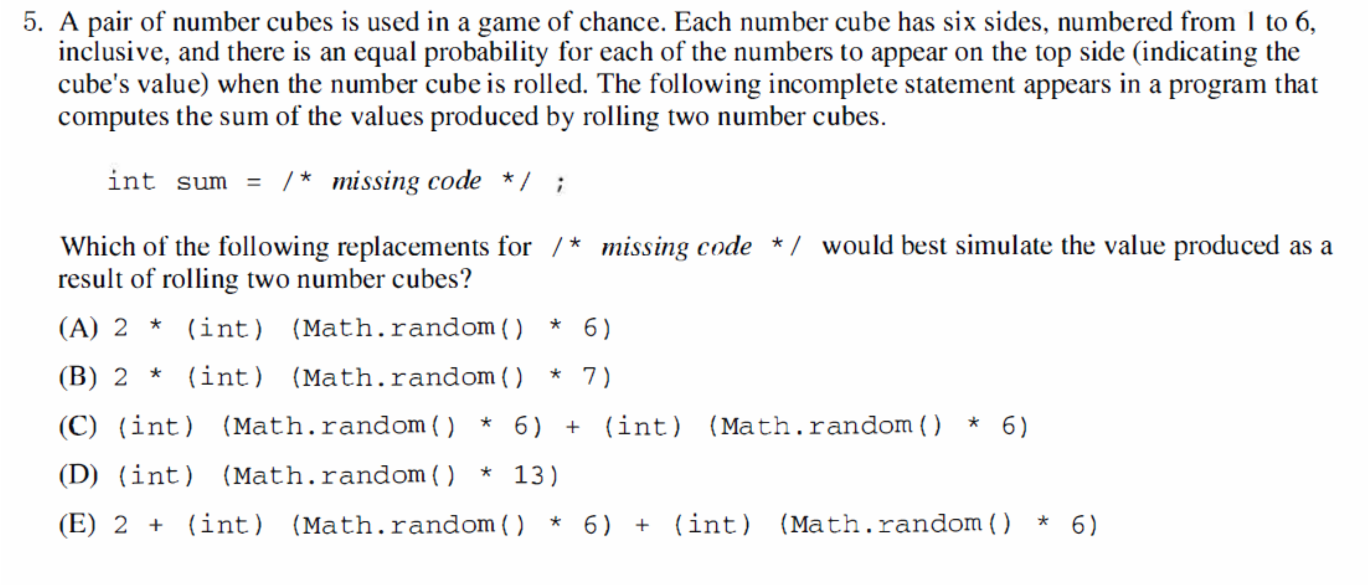
(A) `val - row[num] < minDiff`

(B) `Math.abs(num - minDiff) < minDiff`

(C) `val - num < 0.0`

(D) `Math.abs(num - val) < minDiff`

(E) `Math.abs(row[num] - val) < minDiff`



5. A pair of number cubes is used in a game of chance. Each number cube has six sides, numbered from 1 to 6, inclusive, and there is an equal probability for each of the numbers to appear on the top side (indicating the cube's value) when the number cube is rolled. The following incomplete statement appears in a program that computes the sum of the values produced by rolling two number cubes.

```
int sum = /* missing code */ ;
```

Which of the following replacements for `/*` *missing code* `*/` would best simulate the value produced as a result of rolling two number cubes?

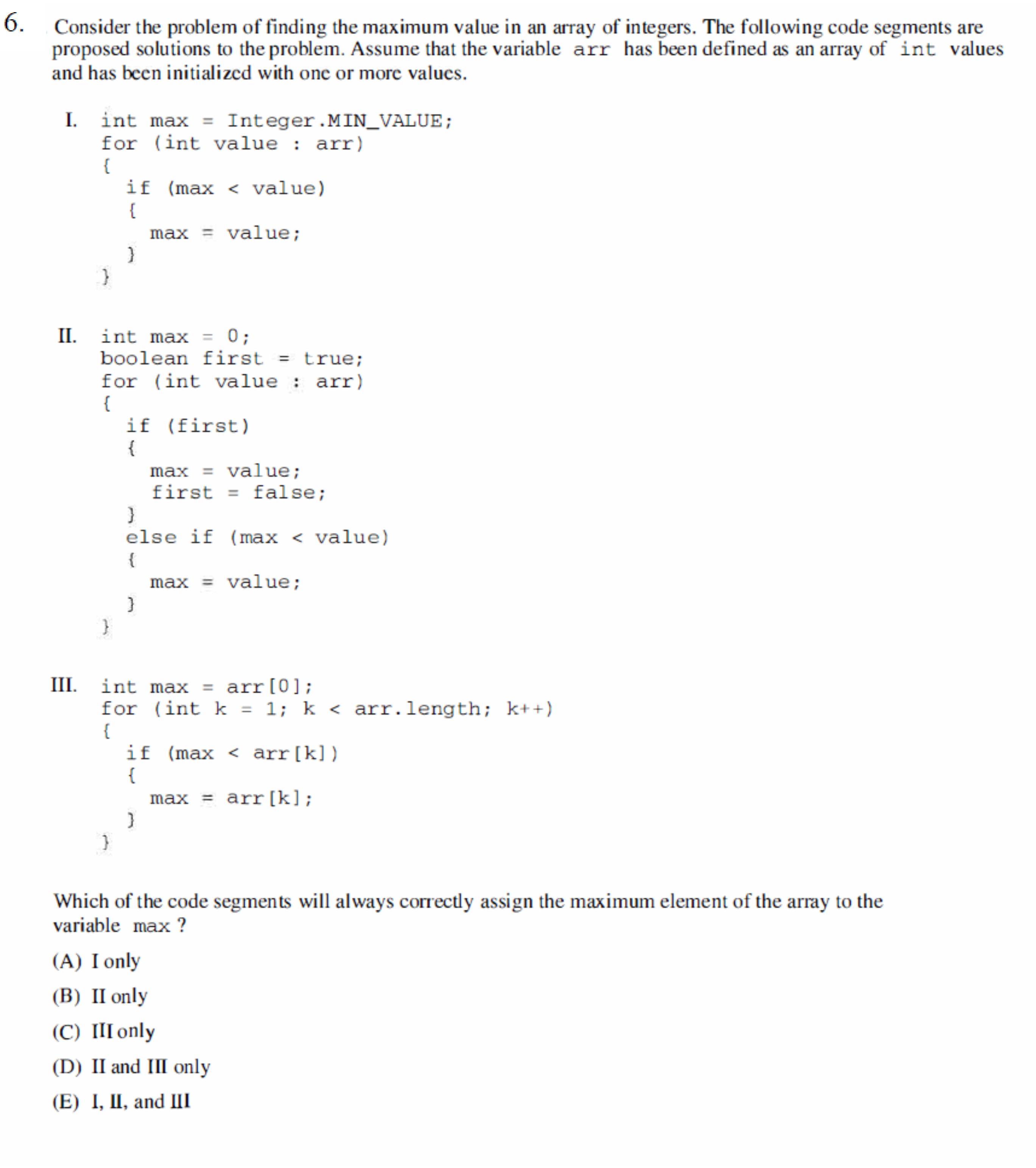
(A) `2 * (int) (Math.random() * 6)`

(B) `2 * (int) (Math.random() * 7)`

(C) `(int) (Math.random() * 6) + (int) (Math.random() * 6)`

(D) `(int) (Math.random() * 13)`

(E) `2 + (int) (Math.random() * 6) + (int) (Math.random() * 6)`



6. Consider the problem of finding the maximum value in an array of integers. The following code segments are proposed solutions to the problem. Assume that the variable `arr` has been defined as an array of `int` values and has been initialized with one or more values.

I.
```
int max = Integer.MIN_VALUE;
for (int value : arr)
{
  if (max < value)
  {
    max = value;
  }
}
```

II.
```
int max = 0;
boolean first = true;
for (int value : arr)
{
  if (first)
  {
    max = value;
    first = false;
  }
  else if (max < value)
  {
    max = value;
  }
}
```

III.
```
int max = arr[0];
for (int k = 1; k < arr.length; k++)
{
  if (max < arr[k])
  {
    max = arr[k];
  }
}
```

Which of the code segments will always correctly assign the maximum element of the array to the variable `max` ?

(A) I only

(B) II only

(C) III only

(D) II and III only

(E) I, II, and III

Class Multiple-Choice

1. Consider the following class definition.

```
public class Bird
{
   private String species;
   private String color;
   private boolean canFly;

   public Bird(String str, String col, boolean cf)
   {
      species = str;
      color = col;
      canFly = cf;
   }
}
```

Which of the following constructors, if added to the `Bird` class, will cause a compilation error?

(A)
```
public Bird()
{
   species = "unknown";
   color = "unknown";
   canFly = false;
}
```

(B)
```
public Bird(boolean cf)
{
   species = "unknown";
   color = "unknown";
   canFly = cf;
}
```

(C)
```
public Bird(String col, String str)
{
   species = str;
   color = col;
   canFly = false;
}
```

(D)

```
public Bird(boolean cf, String str, String col)
{
   species = str;
   color = col;
   canFly = cf;
}
```

(E)

```
public Bird(String col, String str, boolean cf)
{
   species = str;
   color = col;
   canFly = cf;
}
```

2. Consider the following class definitions.

```
public class Person
{
   private String name;

   public String getName()
   {  return name;  }
}

public class Book
{
   private String author;
   private String title;
   private Person borrower;

   public Book(String a, String t)
   {
      author = a;
      title = t;
      borrower = null;
   }

   public void printDetails()
   {
      System.out.print("Author: " + author + " Title: " + title);

      if ( /* missing condition */ )
      {
         System.out.println(" Borrower: " + borrower.getName());
      }
   }

   public void setBorrower(Person b)
   {  borrower = b;  }
}
```

Which of the following can replace `/*` *missing condition* `*/` so that the `printDetails` method CANNOT cause a run-time error?

I. `!borrower.equals(null)`

II. `borrower != null`

III. `borrower.getName() != null`

(A) I only

(B) II only

(C) III only

(D) I and II

(E) II and III

3. Consider the following class definition.

```
public class Points
{
   private double num1;
   private double num2;

   public Points(int n1, int n2)            // Line 6
   {
      num1 = n1;                            // Line 8
      num2 = n2;                            // Line 9
   }

   public void incrementPoints(int value)   // Line 12
   {
      n1 += value;                          // Line 14
      n2 += value;                          // Line 15
   }
}
```

The class does not compile. Which of the following identifies the error in the class definition?

(A) In line 6, the `Points` constructor must have a `void` return type.

(B) In lines 8 and 9, `int` values cannot be assigned to `double` variables.

(C) In line 12, the `incrementPoints` method must have a non-`void` return type.

(D) In lines 14 and 15, the variables `n1` and `n2` are not defined.

(E) In lines 14 and 15, the variable `value` is not defined.

4. Consider the following class definition.

```
public class Book
{
   private int pages;

   public int getPages()
   {
      return pages;
   }

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

The following code segment is intended to store in `maxPages` the greatest number of pages found in any `Book` object in the array `bookArr`.

```
Book[] bookArr = { /* initial values not shown */ };
int maxPages = bookArr[0].getPages();

for (Book b : bookArr)
{
   /* missing code */
}
```

Which of the following can replace `/*` *missing code* `*/` so the code segment works as intended?

(A)
```
if (b.pages > maxPages)
{
   maxPages = b.pages;
}
```

(B)
```
if (b.getPages() > maxPages)
{
   maxPages = b.getPages();
}
```

(C)
```
if (Book[b].pages > maxPages)
{
   maxPages = Book[b].pages;
}
```

(D)
```
if (bookArr[b].pages > maxPages)
{
   maxPages = bookArr[b].pages;
}
```

(E)
```
if (bookArr[b].getPages() > maxPages)
{
   maxPages = bookArr[b].getPages();
}
```

5. Consider the following class definition.

```
public class SomeClass
{
   private int x = 0;
   private static int y = 0;

   public SomeClass(int pX)
   {
      x = pX;
      y++;
   }

   public void incrementY()
   {   y++;  }

   public void incrementY(int inc)
   {   y += inc;  }

   public int getY()
   {   return y;  }
}
```

The following code segment appears in a class other than `SomeClass`.

```
SomeClass first = new SomeClass(10);
SomeClass second = new SomeClass(20);
SomeClass third = new SomeClass(30);
first.incrementY();
second.incrementY(10);
System.out.println(third.getY());
```

What is printed as a result of executing the code segment if the code segment is the first use of a `SomeClass` object?

(A) `0`

(B) `1`

(C) `11`

(D) `14`

(E) `30`

Questions 6-7 refer to the following information.

Consider the following partial class declaration.

```
public class SomeClass
{
  private int myA;
  private int myB;
  private int myC;

  // Constructor(s) not shown

  public int getA()
  {  return myA;  }

  public void setB(int value)
  {  myB = value;  }
}
```

6. The following declaration appears in another class.

```
SomeClass obj = new SomeClass();
```

Which of the following code segments will compile without error?

(A) `int x = obj.getA();`

(B)
```
int x;
obj.getA(x);
```

(C) `int x = obj.myA;`

(D) `int x = SomeClass.getA();`

(E) `int x = getA(obj);`

---

7. Which of the following changes to `SomeClass` will allow other classes to access but not modify the value of `myC` ?

(A) Make `myC` public.

(B) Include the method:
```
public int getC()
 {  return myC;  }
```

(C) Include the method:
```
private int getC()
{  return myC;  }
```

(D) Include the method:
```
public void getC(int x)
{  x = myC;  }
```

(E) Include the method:
```
private void getC(int x)
{  x = myC;  }
```

8. Consider the following class declaration.

```
public class Person
{
  private String myName;
  private int myYearOfBirth;

  public Person(String name, int yearOfBirth)
  {
    myName = name;
    myYearOfBirth = yearOfBirth;
  }

  public String getName()
  {  return myName;  }

  public void setName(String name)
  {  myName = name;  }

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

Assume that the following declaration has been made.

```
Person student = new Person("Thomas", 1995);
```

Which of the following statements is the most appropriate for changing the name of `student` from `"Thomas"` to `"Tom"` ?

(A) `student = new Person("Tom", 1995);`

(B) `student.myName = "Tom";`

(C) `student.getName("Tom");`

(D) `student.setName("Tom");`

(E) `Person.setName("Tom");`

9. Consider the following class declaration.

```
public class Student
{
  private String myName;
  private int myAge;

  public Student()
  {  /* implementation not shown */  }

  public Student(String name, int age)
  {  /* implementation not shown */  }

  // No other constructors
}
```

Which of the following declarations will compile without error?

I. `Student a = new Student();`

II. `Student b = new Student("Juan", 15);`

III. `Student c = new Student("Juan", "15");`

(A) I only

(B) II only

(C) I and II only

(D) I and III only

(E) I, II, and III

10. Consider the following class declarations.

```
public class Point
{
  private double x;   // x-coordinate
  private double y;   // y-coordinate

  public Point()
  {
    x = 0;
    y = 0;
  }

  public Point(double a, double b)
  {
    x = a;
    y = b;
  }

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


public class Circle
{
  private Point center;
  private double radius;

  /** Constructs a circle where (a, b) is the center and r is the radius.
   */
  public Circle(double a, double b, double r)
  {
    /* missing code */
  }
}
```

Which of the following replacements for `/* missing code */` will correctly implement the `Circle` constructor?

I.
```
center = new Point();
radius = r;
```

II.
```
center = new Point(a, b);
radius = r;
```

III.
```
center = new Point();
center.x = a;
center.y = b;
radius = r;
```

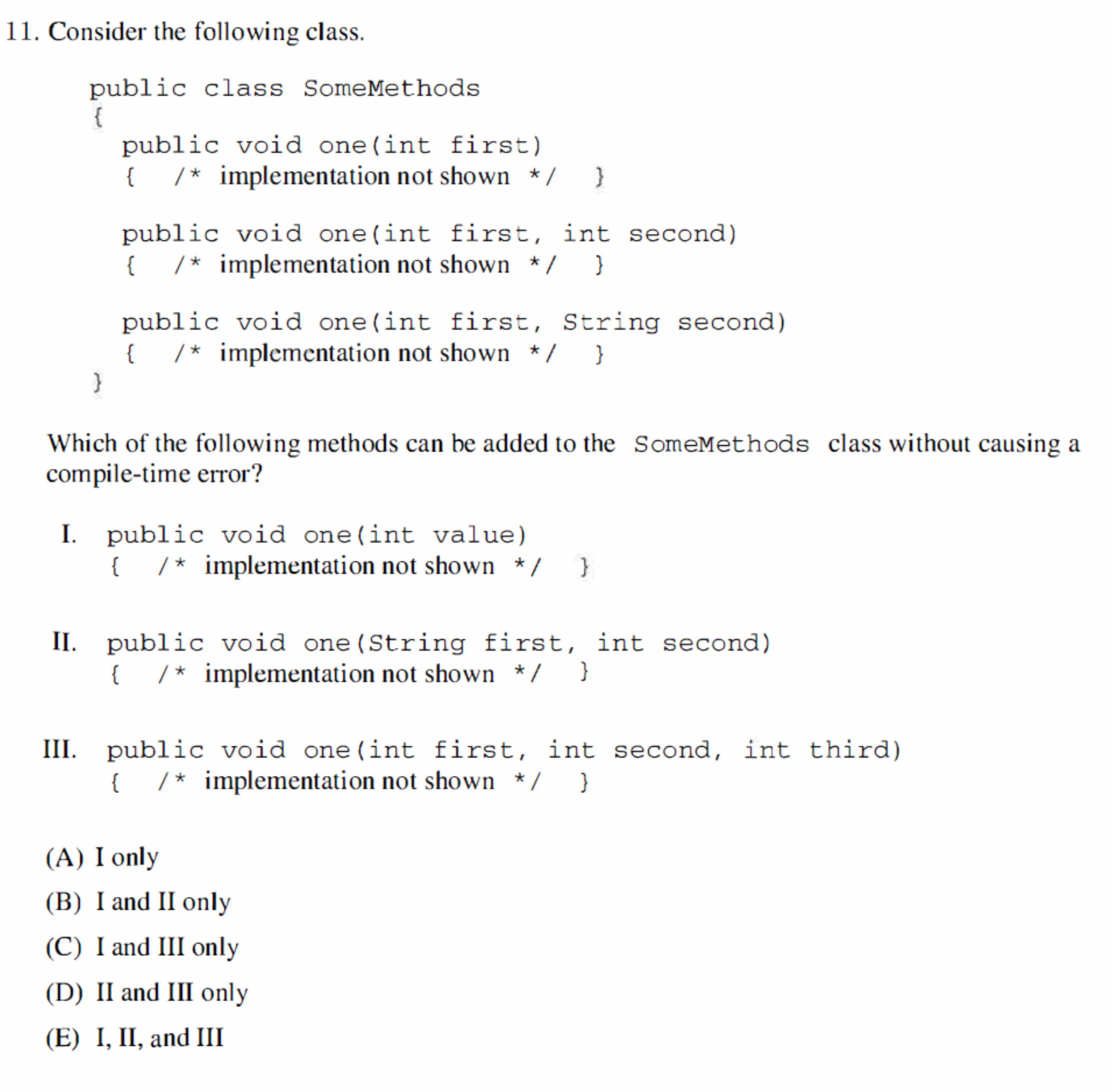
(A) I only

(B) II only

(C) III only

(D) II and III only

(E) I, II, and III



11. Consider the following class.

```
public class SomeMethods
{
  public void one(int first)
  {   /* implementation not shown */   }

  public void one(int first, int second)
  {   /* implementation not shown */   }

  public void one(int first, String second)
  {   /* implementation not shown */   }
}
```

Which of the following methods can be added to the `SomeMethods` class without causing a compile-time error?

I.
```
public void one(int value)
{   /* implementation not shown */   }
```

II.
```
public void one(String first, int second)
{   /* implementation not shown */   }
```

III.
```
public void one(int first, int second, int third)
{   /* implementation not shown */   }
```

(A) I only

(B) I and II only

(C) I and III only

(D) II and III only

(E) I, II, and III

ArrayList Multiple-Choice

1. Consider the following code segment.

```
ArrayList<Integer> numList = new ArrayList<Integer>();

numList.add(3);
numList.add(2);
numList.add(1);
numList.add(1, 0);
numList.set(0, 2);

System.out.print(numList);
```

What is printed by the code segment?

(A) `[1, 3, 0, 1]`

(B) `[2, 0, 2, 1]`

(C) `[2, 0, 2, 3]`

(D) `[2, 3, 2, 1]`

(E) `[3, 0, 0, 1]`

2. The following method is intended to remove all elements of an ArrayList of integers that are divisible by key and add the removed elements to a new ArrayList, which the method returns.

```
public static ArrayList<Integer> match(ArrayList<Integer> numList, int key)
{
   ArrayList<Integer> returnList = new ArrayList<Integer>();

   int i = 0;
   while (i < numList.size())
   {
      int num = numList.get(i);
      if (num % key == 0)
      {
         numList.remove(i);
         returnList.add(num);
      }
      i++;
   }
   return returnList;
}
```

As an example, if the method is called with an ArrayList containing the values [5, 2, 10, 20, 16] and the parameter key has the value 5, then numList should contain [2, 16] at the end of the method and an ArrayList containing [5, 10, 20] should be returned.

Which of the following best explains why the method does not always work as intended?

(A) The method attempts to add an element to returnList after that element has already been removed from numList.

(B) The method causes a NullPointerException to be thrown when no matches are found.

(C) The method causes an IndexOutOfBoundsException to be thrown.

(D) The method fails to correctly determine whether an element of numList is divisible by key.

(E) The method skips some elements of numList during the traversal.

3. Consider the following class definition.

```
public class Value
{
   private int num;

   public int getNum()
   {
      return num;
   }

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

The following method appears in a class other than `Value`. It is intended to sum all the `num` instance variables of the `Value` objects in its `ArrayList` parameter.

```
/** Precondition: valueList is not null */
public static int getTotal(ArrayList<Value> valueList)
{
   int total = 0;
   /* missing code */
   return total;
}
```

Which of the following code segments can replace `/*` *missing code* `*/` so the `getTotal` method works as intended?

```
  I. for (int x = 0; x < valueList.size(); x++)
     {
        total += valueList.get(x).getNum();
     }

 II. for (Value v : valueList)
     {
        total += v.getNum();
     }

III. for (Value v : valueList)
     {
        total += getNum(v);
     }
```

(A) I only

(B) II only

(C) III only

(D) I and II

(E) I and III

4. Consider the following method.

```
public ArrayList<Integer> mystery(int n)
{
  ArrayList<Integer> seq = new ArrayList<Integer>();

  for (int k = 1; k <= n; k++)
    seq.add(new Integer(k * k + 3));

  return seq;
}
```

Which of the following is printed as a result of executing the following statement?

```
System.out.println(mystery(6));
```

(A) `[3, 4, 7, 12, 19, 28]`

(B) `[3, 4, 7, 12, 19, 28, 39]`

(C) `[4, 7, 12, 19, 28, 39]`

(D) `[39, 28, 19, 12, 7, 4]`

(E) `[39, 28, 19, 12, 7, 4, 3]`

5. Consider the following class and class declarations.

```
public class Vehicle  {

  /** @return  the mileage traveled by this  Vehicle
   */
  double getMileage();
}


public class Fleet
{
  private ArrayList<Vehicle> myVehicles;

  /** @return  the mileage traveled by all vehicles in this  Fleet
   */
  public double getTotalMileage()
  {
    double sum = 0.0;

    for (Vehicle v : myVehicles)
    {
      sum += /* expression */ ;
    }

    return sum;
  }

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

Which of the following can be used to replace `/* expression */` so that `getTotalMileage` returns the total of the miles traveled for all vehicles in the fleet?

(A) `getMileage(v)`

(B) `myVehicles[v].getMileage()`

(C) `Vehicle.get(v).getMileage()`

(D) `myVehicles.get(v).getMileage()`

(E) `v.getMileage()`

6. Assume that `myList` is an `ArrayList` that has been correctly constructed and populated with objects. Which of the following expressions produces a valid random index for `myList` ?

(A) `(int)( Math.random() * myList.size() ) - 1`

(B) `(int)( Math.random() * myList.size() )`

(C) `(int)( Math.random() * myList.size() ) + 1`

(D) `(int)( Math.random() * (myList.size() + 1) )`

(E) `Math.random(myList.size())`

7. Consider the following instance variable and method.

```
private ArrayList<String> animals;

public void manipulate()
{
  for (int k = animals.size() - 1; k > 0; k--)
  {
    if (animals.get(k).substring(0, 1).equals("b"))
    {
      animals.add(animals.size() - k, animals.remove(k));
    }
  }
}
```

Assume that `animals` has been instantiated and initialized with the following contents.

```
["bear", "zebra", "bass", "cat", "koala", "baboon"]
```

What will the contents of `animals` be as a result of calling `manipulate` ?

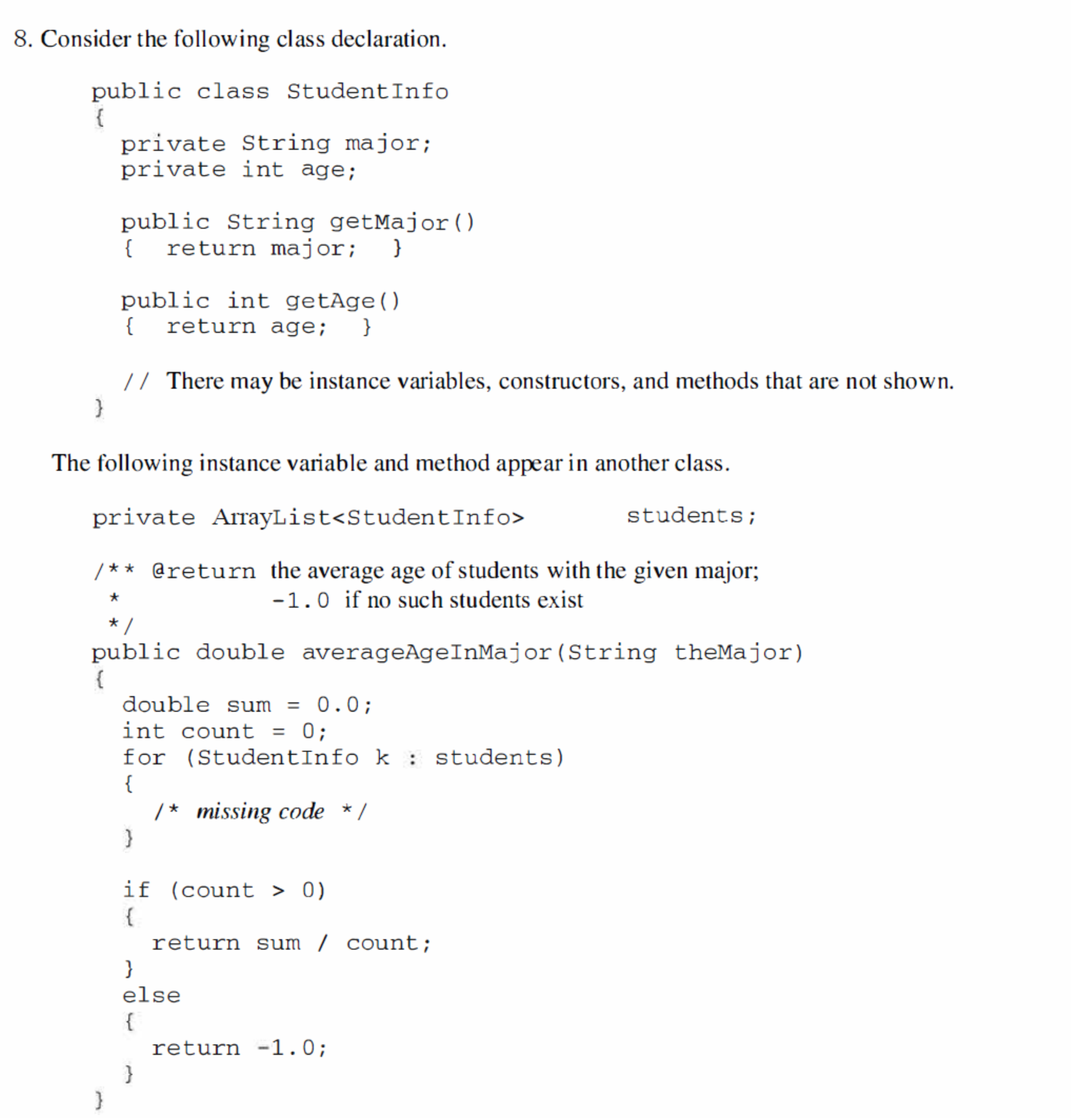
(A) `["baboon", "zebra", "bass", "cat", "bear", "koala"]`

(B) `["bear", "zebra", "bass", "cat", "koala", "baboon"]`

(C) `["baboon", "bear", "zebra", "bass", "cat", "koala"]`

(D) `["bear", "baboon", "zebra", "bass", "cat", "koala"]`

(E) `["zebra", "cat", "koala", "baboon", "bass", "bear"]`



8. Consider the following class declaration.

```
public class StudentInfo
{
  private String major;
  private int age;

  public String getMajor()
  {   return major;   }

  public int getAge()
  {   return age;   }

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

The following instance variable and method appear in another class.

```
private ArrayList<StudentInfo>       students;

/** @return the average age of students with the given major;
 *          -1.0 if no such students exist
 */
public double averageAgeInMajor(String theMajor)
{
  double sum = 0.0;
  int count = 0;
  for (StudentInfo k : students)
  {
    /* missing code */
  }

  if (count > 0)
  {
    return sum / count;
  }
  else
  {
    return -1.0;
  }
}
```

Which of the following could be used to replace `/* missing code */` so that `averageAgeInMajor` will compile without error?

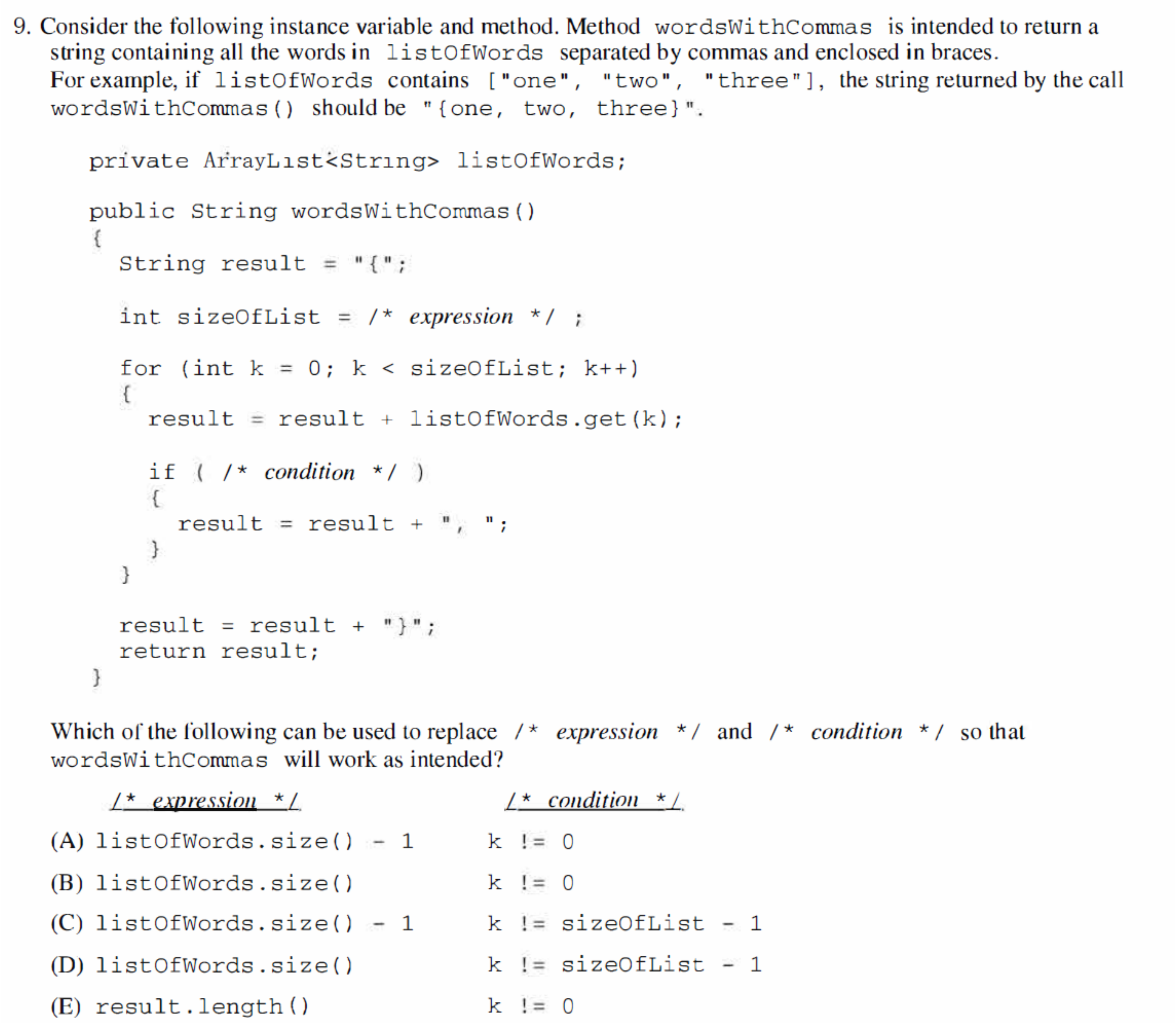
(A)
```
if (theMajor.equals(k.major))
{
  sum += k.age;
  count++;
}
```

(B)
```
if (theMajor.equals(k.getMajor()))
{
  sum += k.getAge();
  count++;
}
```

(C)
```
if (theMajor.equals(k.major))
{
  sum += k.getAge();
  count++;
}
```

(D)
```
if (theMajor.equals(students[k].getMajor()))
{
  sum += students[k].getAge();
  count++;
}
```

(E)
```
if (theMajor.equals(getMajor(k)))
{
  sum += getAge(k);
  count++;
}
```

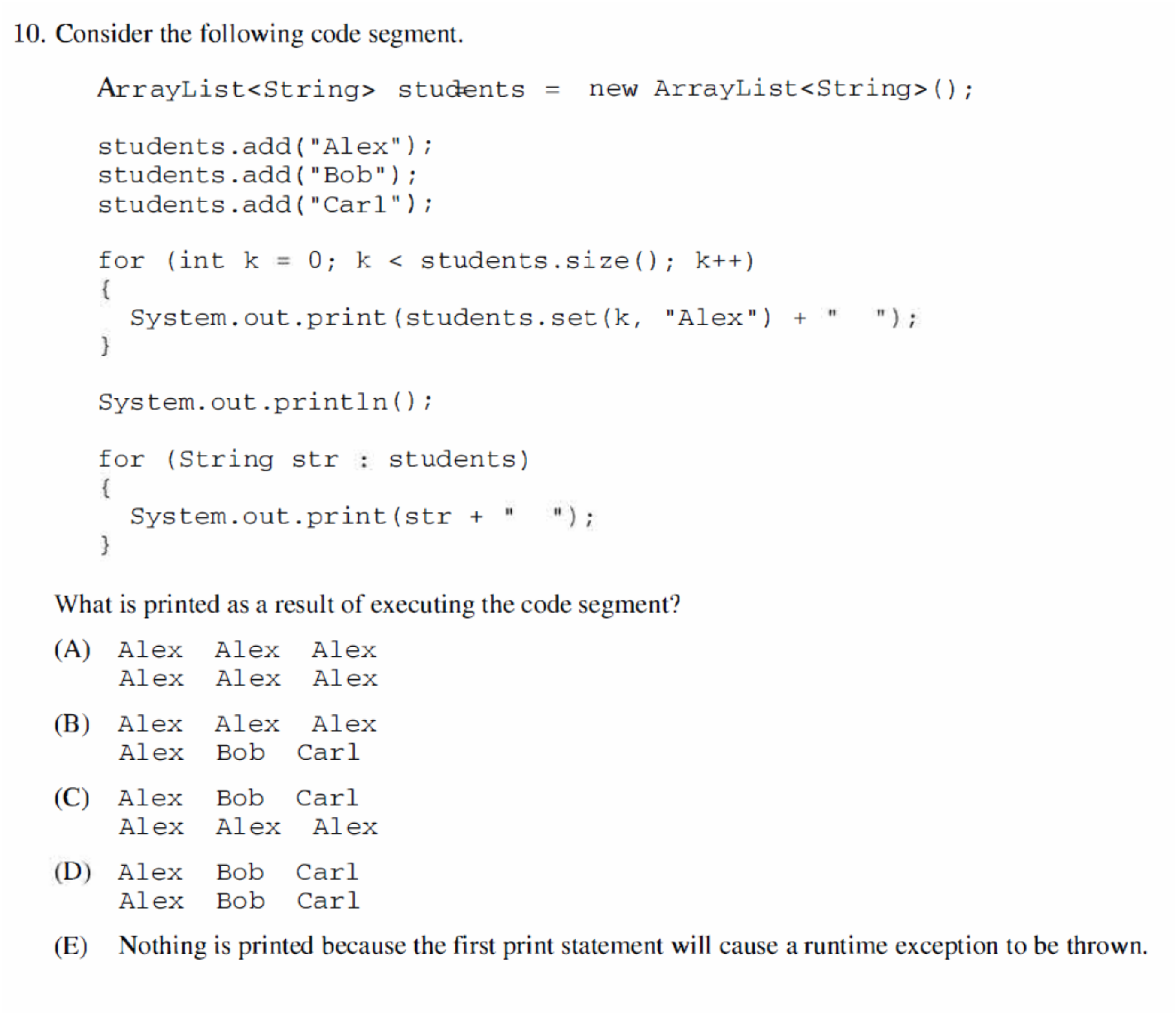


9. Consider the following instance variable and method. Method `wordsWithCommas` is intended to return a string containing all the words in `listOfWords` separated by commas and enclosed in braces.
For example, if `listOfWords` contains `["one", "two", "three"]`, the string returned by the call `wordsWithCommas()` should be `"{one, two, three}"`.

```
private ArrayList<String> listOfWords;

public String wordsWithCommas()
{
  String result = "{";

  int sizeOfList = /* expression */ ;

  for (int k = 0; k < sizeOfList; k++)
  {
    result = result + listOfWords.get(k);

    if ( /* condition */ )
    {
      result = result + ", ";
    }
  }

  result = result + "}";
  return result;
}
```

Which of the following can be used to replace `/* expression */` and `/* condition */` so that `wordsWithCommas` will work as intended?

	/ expression */*	*/* condition */*
(A)	`listOfWords.size() - 1`	`k != 0`
(B)	`listOfWords.size()`	`k != 0`
(C)	`listOfWords.size() - 1`	`k != sizeOfList - 1`
(D)	`listOfWords.size()`	`k != sizeOfList - 1`
(E)	`result.length()`	`k != 0`



10. Consider the following code segment.

```
ArrayList<String> students =  new ArrayList<String>();

students.add("Alex");
students.add("Bob");
students.add("Carl");

for (int k = 0; k < students.size(); k++)
{
  System.out.print(students.set(k, "Alex") + "  ");
}

System.out.println();

for (String str : students)
{
  System.out.print(str + "  ");
}
```

What is printed as a result of executing the code segment?

(A)
```
Alex  Alex  Alex
Alex  Alex  Alex
```

(B)
```
Alex  Alex  Alex
Alex  Bob  Carl
```

(C)
```
Alex  Bob  Carl
Alex  Alex  Alex
```

(D)
```
Alex  Bob  Carl
Alex  Bob  Carl
```

(E) Nothing is printed because the first print statement will cause a runtime exception to be thrown.

Inheritance Multiple-choice

1. Consider the following class definitions.

```
public class Hero
{
   private String name;
   private int power;

   public Hero(String n, int p)
   {
      name = n;
      power = p;
   }

   public void powerUp(int p)
   {
      power += p;
   }

   public int showPower()
   {   return power;   }
}

public class SuperHero extends Hero
{
   public SuperHero(String n, int p)
   {
      super(n, p);
   }

   public void powerUp(int p)
   {
      super.powerUp(p * 2);
   }
}
```

The following code segment appears in a class other than `Hero` and `SuperHero`.

```
Hero j = new SuperHero("JavaHero", 50);
j.powerUp(10);
System.out.println(j.showPower());
```

What is printed as a result of executing the code segment?

(A) `10`

(B) `20`

(C) `60`

(D) `70`

(E) `100`

2. Consider the following class definitions.

```
public class Rectangle
{
   private int height;
   private int width;

   public Rectangle()
   {
      height = 1;
      width = 1;
   }

   public Rectangle(int x)
   {
      height = x;
      width = x;
   }

   public Rectangle(int h, int w)
   {
      height = h;
      width = w;
   }

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

public class Square extends Rectangle
{
   public Square(int x)
   {
      /* missing code */
   }
}
```

Which of the following code segments can replace `/* missing code */` so that the `Square` class constructor initializes the `Rectangle` class instance variables `height` and `width` to `x` ?

(A) `super();`

(B) `super(x);`

(C) `Rectangle(x);`

(D) `Square(x, x);`

(E)
```
height = x;
width = x;
```

3. Consider the following class definitions.

```
public class ClassA
{
   public String getValue()
   {
      return "A";
   }

   public void showValue()
   {
      System.out.print(getValue());
   }
}

public class ClassB extends ClassA
{
   public String getValue()
   {
      return "B";
   }
}
```

The following code segment appears in a class other than `ClassA` or `ClassB`.

```
ClassA obj = new ClassB();
obj.showValue();
```

What, if anything, is printed when the code segment is executed?

(A) `A`

(B) `B`

(C) `AB`

(D) `BA`

(E) Nothing is printed because the code does not compile.

4. Consider the following class definitions.

```
public class A
{
   public String message(int i)
   {
      return "A" + i;
   }
}

public class B extends A
{
   public String message(int i)
   {
      return "B" + i;
   }
}
```

The following code segment appears in a class other than A or B.

```
A obj1 = new B();                        // Line 1
B obj2 = new B();                        // Line 2
System.out.println(obj1.message(3));   // Line 3
System.out.println(obj2.message(2));   // Line 4
```

Which of the following best explains the difference, if any, in the behavior of the code segment that will result from removing the `message` method from class `A` ?

(A) The statement in line 3 will cause a compiler error because the `message` method for `obj1` cannot be found.

(B) The statement in line 4 will cause a compiler error because the `message` method for `obj2` cannot be found.

(C) As a result of the method call in line 3, the `message` method in class `B` will be executed instead of the `message` method in class `A`.

(D) As a result of the method call in line 4, the `message` method in class `B` will be executed instead of the `message` method in class `A`.

(E) The behavior of the code segment will remain unchanged.

5. Consider the following `Book` and `AudioBook` classes.

```
public class Book
{
  private int numPages;
  private String bookTitle;

  public Book(int pages, String title)
  {
    numPages = pages;
    bookTitle = title;
  }

  public String toString()
  {
    return bookTitle + " " + numPages;
  }

  public int length()
  {
    return numPages;
  }
}


public class AudioBook extends Book
{
  private int numMinutes;

  public AudioBook(int minutes, int pages, String title)
  {
    super(pages, title);
    numMinutes = minutes;
  }

  public int length()
  {
    return numMinutes;
  }

  public double pagesPerMinute()
  {
    return ((double) super.length()) / numMinutes;
  }
}
```

Consider the following code segment that appears in a class other than `Book` or `AudioBook`.

```
Line 1:   Book[] books = new Book[2];
Line 2:   books[0] = new AudioBook(100, 300, "The Jungle");
Line 3:   books[1] = new Book(400, "Captains Courageous");
Line 4:   System.out.println(books[0].pagesPerMinute());
Line 5:   System.out.println(books[0].toString());
Line 6:   System.out.println(books[0].length());
Line 7:   System.out.println(books[1].toString());
```

Which of the following best explains why the code segment will not compile?

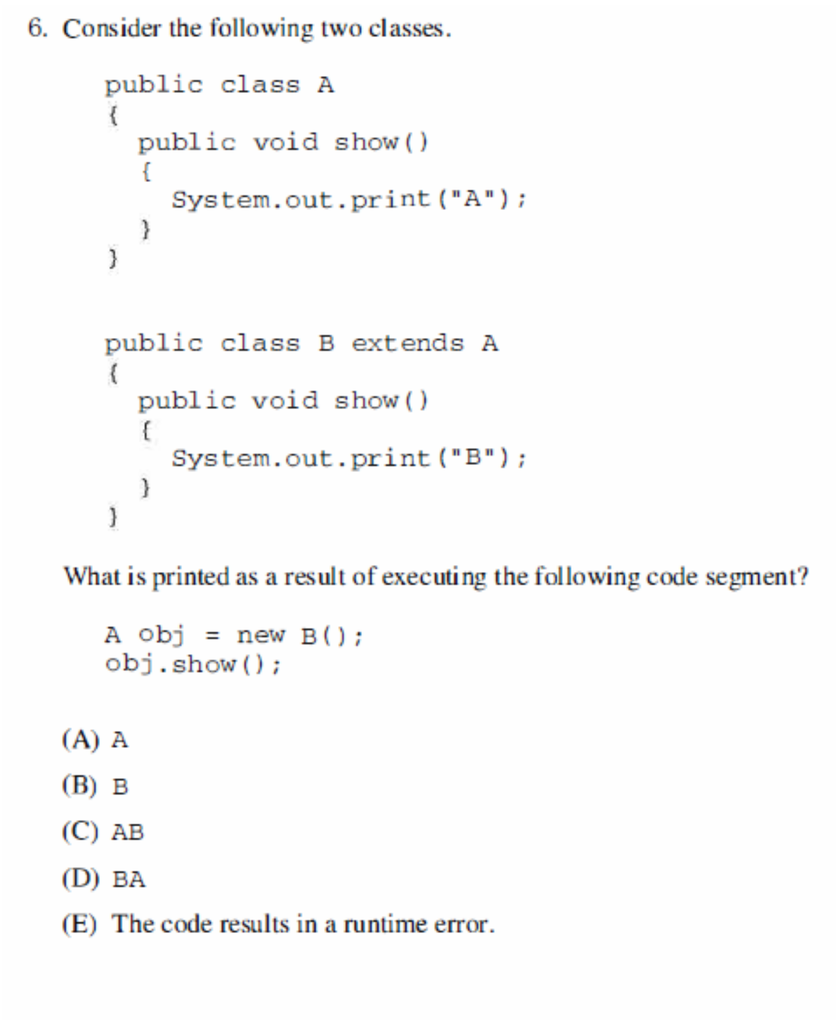
(A) Line 2 will not compile because variables of type `Book` may not refer to variables of type `AudioBook`.

(B) Line 4 will not compile because variables of type `Book` may only call methods in the `Book` class.

(C) Line 5 will not compile because the `AudioBook` class does not have a method named `toString` declared or implemented.

(D) Line 6 will not compile because the statement is ambiguous. The compiler cannot determine which `length` method should be called.

(E) Line 7 will not compile because the element at index 1 in the array named `books` may not have been initialized.



6. Consider the following two classes.

```
public class A
{
  public void show()
  {
    System.out.print("A");
  }
}


public class B extends A
{
  public void show()
  {
    System.out.print("B");
  }
}
```

What is printed as a result of executing the following code segment?

```
A obj = new B();
obj.show();
```

(A) `A`

(B) `B`

(C) `AB`

(D) `BA`

(E) The code results in a runtime error.

Recursion Multiple-choice

1. Consider the following recursive method.

```
public static void stars(int num)
{
   if (num == 1)
   {
      return;
   }

   stars(num - 1);

   for (int i = 0; i < num; i++)
   {
      System.out.print("*");
   }
   System.out.println();
}
```

What is printed as a result of the method call `stars(5)` ?

(A)

```
*****
```

(B)

```
**
***
****
*****
```

(C)

```
*
**
***
****
*****
```

(D)

```
*****
****
***
**
```

(E)

```
*****
****
***
**
*
```

2. Consider the following method.

```
public static int calcMethod(int num)
{
   if (num == 0)
   {
      return 10;
   }
   return num + calcMethod(num / 2);
}
```

What value is returned by the method call `calcMethod(16)` ?

(A) `10`

(B) `26`

(C) `31`

(D) `38`

(E) `41`

3. Consider the following recursive method.

```
public static boolean recurMethod(String str)
{
   if (str.length() <= 1)
   {
      return true;
   }
   else if (str.substring(0, 1).compareTo(str.substring(1, 2)) > 0)
   {
      return recurMethod(str.substring(1));
   }
   else
   {
      return false;
   }
}
```

Which of the following method calls will return `true` ?

(A) `recurMethod("abcba")`

(B) `recurMethod("abcde")`

(C) `recurMethod("bcdab")`

(D) `recurMethod("edcba")`

(E) `recurMethod("edcde")`

Questions 4-5 refer to the following information.

Consider the following instance variable and methods. You may assume that `data` has been initialized with length > 0. The methods are intended to return the index of an array element equal to `target`, or -1 if no such element exists.

```
private int[] data;

public int seqSearchRec(int target)
{
  return seqSearchRecHelper(target, data.length - 1);
}


private int seqSearchRecHelper(int target, int last)
{
  // Line 1

  if (data[last] == target)
    return last;
  else
    return seqSearchRecHelper(target, last - 1);
}
```

4. For which of the following test cases will the call `seqSearchRec(5)` always result in an error?

I. `data` contains only one element.

II. `data` does not contain the value 5.

III. `data` contains the value 5 multiple times.

(A) I only

(B) II only

(C) III only

(D) I and II only

(E) I, II, and III

---

5. Which of the following should be used to replace `// Line 1` in `seqSearchRecHelper` so that `seqSearchRec` will work as intended?

(A)
```
if (last <= 0)
  return -1;
```

(B)
```
if (last < 0)
  return -1;
```

(C)
```
if (last < data.length)
  return -1;
```

(D)
```
while (last < data.length)
```

(E)
```
while (last >= 0)
```

6. Consider the following recursive method.

```
public int recur(int n)
{
  if (n <= 10)
    return n * 2;
  else
    return recur(recur(n / 3));
}
```

What value is returned as a result of the call `recur(27)` ?

(A) 8

(B) 9

(C) 12

(D) 16

(E) 18

7. Consider the following recursive method.

```
public static void whatsItDo(String str)
{
  int len = str.length();
  if (len > 1)
  {
    String temp = str.substring(0, len - 1);
    whatsItDo(temp);
    System.out.println(temp);
  }
}
```

What is printed as a result of the call `whatsItDo("WATCH")` ?

(A)
```
WATC
WAT
WA
W
```

(B)
```
WATCH
WATC
WAT
WA
```

(C)
```
W
WA
WAT
WATC
```

(D)
```
W
WA
WAT
WATC
WATCH
```

(E)
```
WATCH
WATC
WAT
WA
W
WA
WAT
WATC
WATCH
```

8. Consider the following method.

```
public static void showMe(int arg)
{
  if (arg < 10)
  {
    showMe(arg + 1);
  }
  else
  {
    System.out.print(arg + " ");
  }
}
```

What will be printed as a result of the call `showMe(0)` ?

(A) `10`

(B) `11`

(C) `0 1 2 3 4 5 6 7 8 9`

(D) `9 8 7 6 5 4 3 2 1 0`

(E) `0 1 2 3 4 5 6 7 8 9 10`

9. Consider the following recursive method.

```
public static void whatsItDo(String str)
{
  int len = str.length();
  if (len > 1)
  {
    String temp = str.substring(0, len - 1);
    System.out.println(temp);
    whatsItDo(temp);
  }
}
```

What is printed as a result of the call `whatsItDo("WATCH")` ?

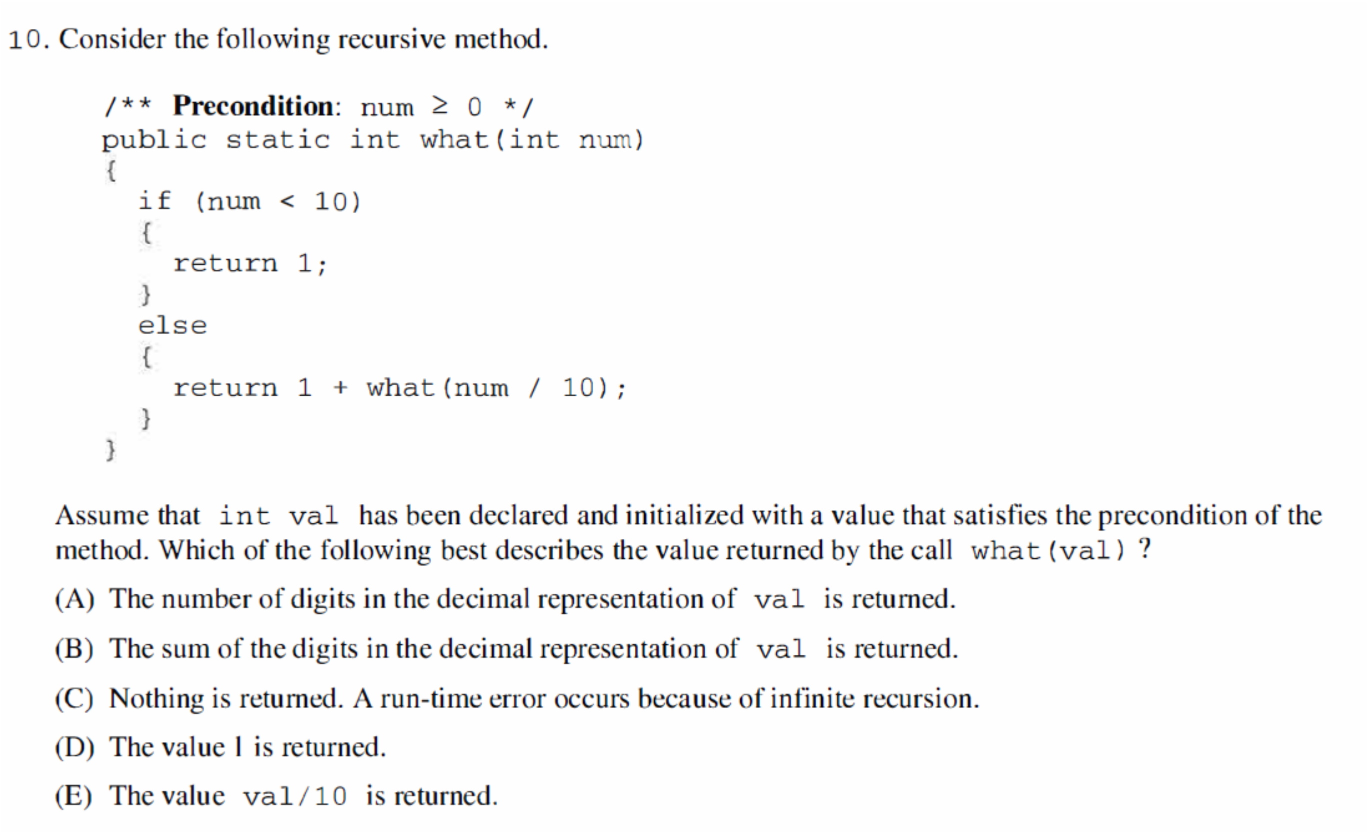
(A)
```
H
```

(B)
```
WATC
```

(C)
```
ATCH
ATC
AT
A
```

(D)
```
WATC
WAT
WA
W
```

(E)
```
WATCH
WATC
WAT
WA
```



10. Consider the following recursive method.

```
/** Precondition: num ≥ 0 */
public static int what(int num)
{
  if (num < 10)
  {
    return 1;
  }
  else
  {
    return 1 + what(num / 10);
  }
}
```

Assume that `int val` has been declared and initialized with a value that satisfies the precondition of the method. Which of the following best describes the value returned by the call `what(val)` ?

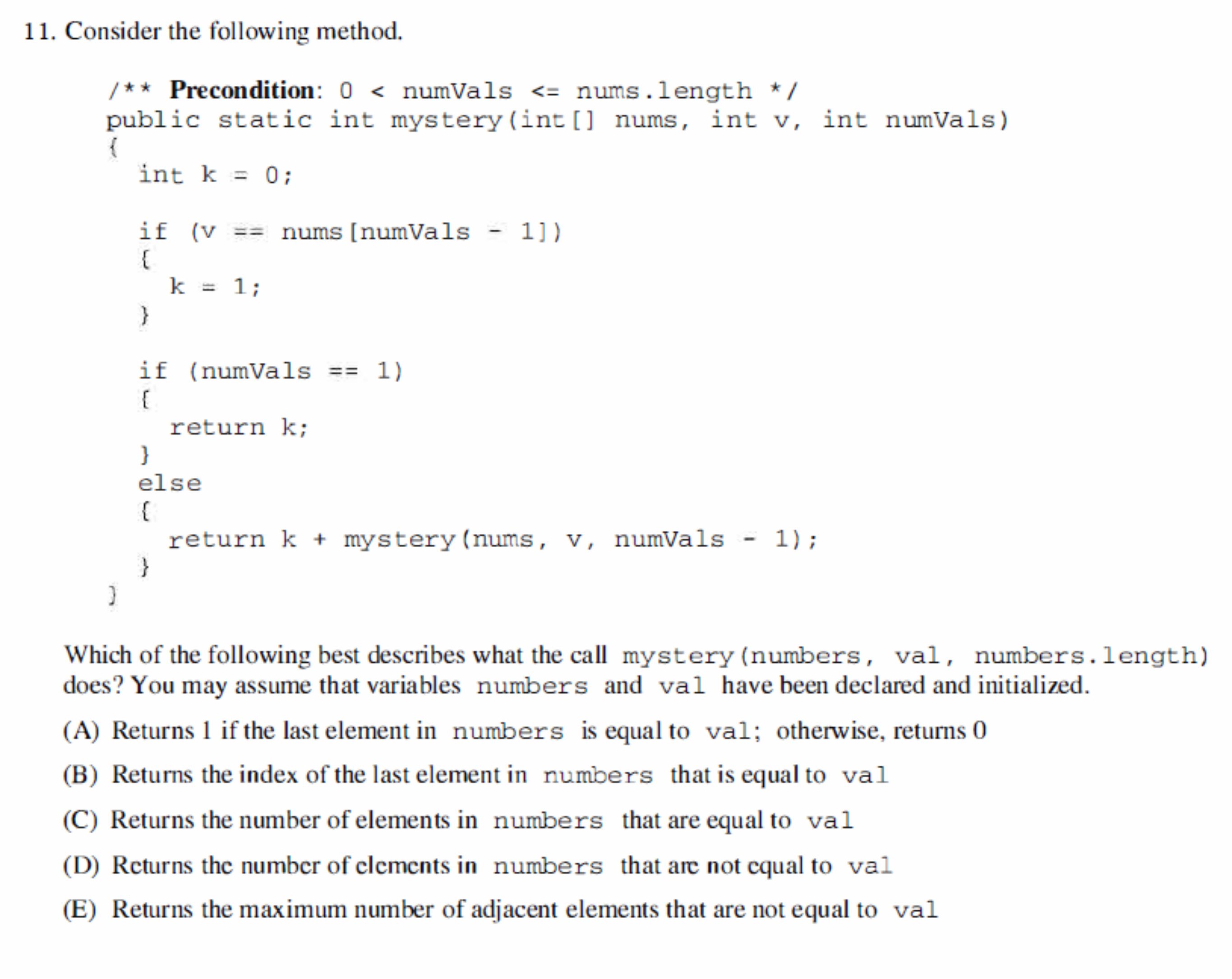
(A) The number of digits in the decimal representation of `val` is returned.

(B) The sum of the digits in the decimal representation of `val` is returned.

(C) Nothing is returned. A run-time error occurs because of infinite recursion.

(D) The value 1 is returned.

(E) The value `val/10` is returned.



11. Consider the following method.

```
/** Precondition: 0 < numVals <= nums.length */
public static int mystery(int[] nums, int v, int numVals)
{
  int k = 0;

  if (v == nums[numVals - 1])
  {
    k = 1;
  }

  if (numVals == 1)
  {
    return k;
  }
  else
  {
    return k + mystery(nums, v, numVals - 1);
  }
}
```

Which of the following best describes what the call `mystery(numbers, val, numbers.length)` does? You may assume that variables `numbers` and `val` have been declared and initialized.

(A) Returns 1 if the last element in `numbers` is equal to `val`; otherwise, returns 0

(B) Returns the index of the last element in `numbers` that is equal to `val`

(C) Returns the number of elements in `numbers` that are equal to `val`

(D) Returns the number of elements in `numbers` that are not equal to `val`

(E) Returns the maximum number of adjacent elements that are not equal to `val`

Sorting and Searching Multiple-choice

Questions 1-2 refer to the following information.

Consider the following `sort` method. This method correctly sorts the elements of array `data` into increasing order.

```
public static void sort(int[] data)
{
  for (int j = 0; j < data.length - 1; j++)
  {
    int m = j;
    for (int k = j + 1; k < data.length; k++)
    {
      if (data[k] < data[m])      /* Compare values */
      {
        m = k;
      }
    }
    int temp = data[m];           /* Assign to temp */
    data[m] = data[j];
    data[j] = temp;

    /* End of outer loop */
  }
}
```

1. Assume that `sort` is called with the array `{6, 3, 2, 5, 4, 1}`. What will the value of `data` be after three passes of the outer loop (i.e., when `j = 2` at the point indicated by `/*` *End of outer loop* `*/`) ?

(A) `{1, 2, 3, 4, 5, 6}`

(B) `{1, 2, 3, 5, 4, 6}`

(C) `{1, 2, 3, 6, 5, 4}`

(D) `{1, 3, 2, 4, 5, 6}`

(E) `{1, 3, 2, 5, 4, 6}`

---

2. Assume that `sort` is called with the array `{1, 2, 3, 4, 5, 6}`. How many times will the expression indicated by `/*` *Compare values* `*/` and the statement indicated by `/*` *Assign to temp* `*/` execute?

	<u>Compare values</u>	<u>Assign to temp</u>
(A)	15	0
(B)	15	5
(C)	15	6
(D)	21	5
(E)	21	6

Questions 3-4 refer to the following information.

Consider the following `binarySearch` method. The method correctly performs a binary search.

```
/** Precondition: data is sorted in increasing order. */
public static int binarySearch(int[] data, int target)
{
  int start = 0;
  int end = data.length - 1;
  while (start <= end)
  {
    int mid = (start + end) / 2;        /* Calculate midpoint */
    if (target < data[mid])
    {
      end = mid - 1;
    }
    else if (target > data[mid])
    {
      start = mid + 1;
    }
    else
    {
      return mid;
    }
  }
  return -1;
}
```

3. Consider the following code segment.

```
int[] values = {1, 2, 3, 4, 5, 8, 8, 8};
int target = 8;
```

What value is returned by the call `binarySearch(values, target)` ?

(A) -1

(B) 3

(C) 5

(D) 6

(E) 8

---

4. Suppose the `binarySearch` method is called with an array containing 2,000 elements sorted in increasing order. What is the maximum number of times that the statement indicated by `/*` *Calculate midpoint* `*/` could execute?

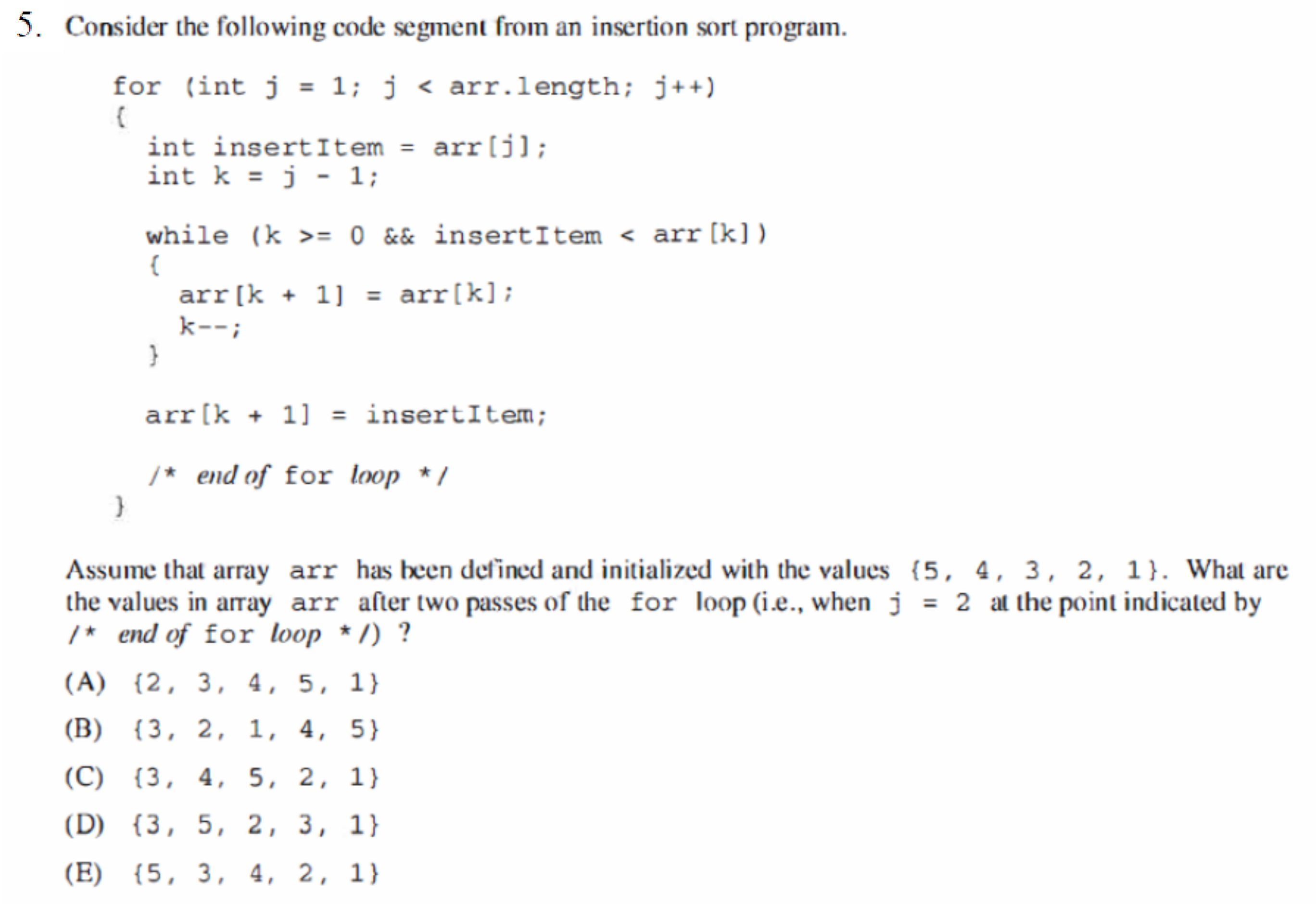
(A) 2,000

(B) 1,000

(C) 20

(D) 11

(E) 1



5. Consider the following code segment from an insertion sort program.

```
for (int j = 1; j < arr.length; j++)
{
  int insertItem = arr[j];
  int k = j - 1;

  while (k >= 0 && insertItem < arr[k])
  {
    arr[k + 1] = arr[k];
    k--;
  }

  arr[k + 1] = insertItem;

  /* end of for loop */
}
```

Assume that array `arr` has been defined and initialized with the values `{5, 4, 3, 2, 1}`. What are the values in array `arr` after two passes of the `for` loop (i.e., when `j = 2` at the point indicated by `/*` *end of* `for` *loop* `*/`) ?

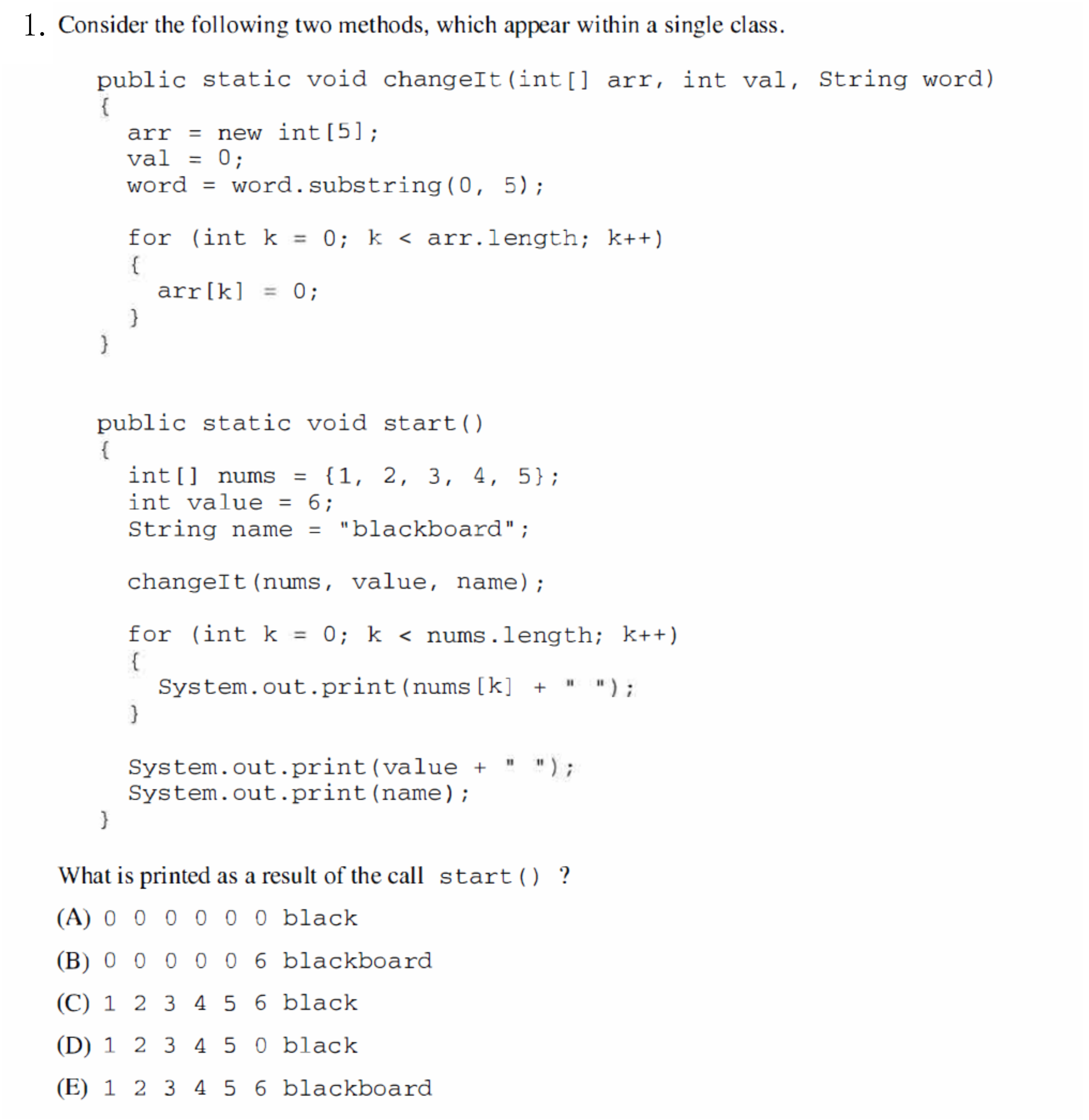
(A) `{2, 3, 4, 5, 1}`

(B) `{3, 2, 1, 4, 5}`

(C) `{3, 4, 5, 2, 1}`

(D) `{3, 5, 2, 3, 1}`

(E) `{5, 3, 4, 2, 1}`

1. 

Consider the following two methods, which appear within a single class.

```
public static void changeIt(int[] arr, int val, String word)
{
  arr = new int[5];
  val = 0;
  word = word.substring(0, 5);

  for (int k = 0; k < arr.length; k++)
  {
    arr[k] = 0;
  }
}


public static void start()
{
  int[] nums = {1, 2, 3, 4, 5};
  int value = 6;
  String name = "blackboard";

  changeIt(nums, value, name);

  for (int k = 0; k < nums.length; k++)
  {
    System.out.print(nums[k] + " ");
  }

  System.out.print(value + " ");
  System.out.print(name);
}
```

What is printed as a result of the call `start()` ?

(A) `0 0 0 0 0 0 black`

(B) `0 0 0 0 0 6 blackboard`

(C) `1 2 3 4 5 6 black`

(D) `1 2 3 4 5 0 black`

(E) `1 2 3 4 5 6 blackboard`

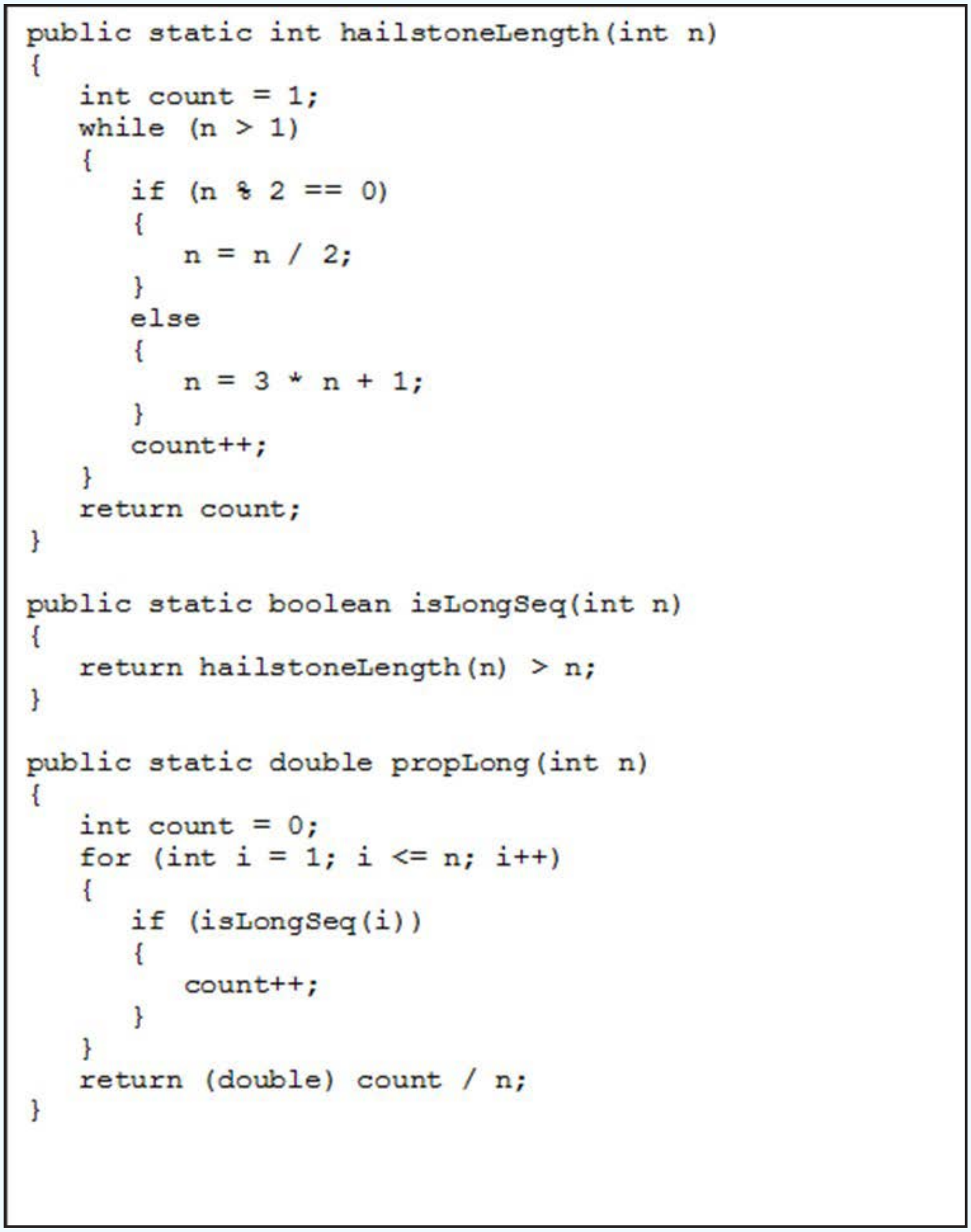
Basic Part FRQ Answer

1. 2019(1).1

Question 1: Methods and Control Structures — 9 points

Learning Objectives: CON-1.A CON-1.C CON-1.E CON-2.A CON-2.C CON-2.E MOD-1.G MOD-2.F

Canonical solution



(a) **3 points**

```
public static int hailstoneLength(int n)
{
   int count = 1;
   while (n > 1)
   {
      if (n % 2 == 0)
      {
         n = n / 2;
      }
      else
      {
         n = 3 * n + 1;
      }
      count++;
   }
   return count;
}
```

(b) **2 points**

```
public static boolean isLongSeq(int n)
{
   return hailstoneLength(n) > n;
}
```

(c) **4 points**

```
public static double propLong(int n)
{
   int count = 0;
   for (int i = 1; i <= n; i++)
   {
      if (isLongSeq(i))
      {
         count++;
      }
   }
   return (double) count / n;
}
```

(a) `hailstoneLength`

	Scoring Criteria	Decision Rules	
1	Loops from given starting value `n` until the sequence terminates, using updated values for the current term	**Responses still earn the point even if they...** • update `n` incorrectly.	**1 point** 3.C CON-2.C
2	Computes the next value	**Responses still earn the point even if they...** • use a correct formula in an incorrect case.	**1 point** 3.C CON-1.A
3	Uses correct formula for next value depending on even/odd		**1 point** 3.C CON-2.A
		Total for part (a)	**3 points**

(b) `isLongSeq`

	Scoring Criteria	Decision Rules	
4	Calls `hailstoneLength`		**1 point** 3.A MOD-1.G
5	Correctly compares length and starting value to determine return value	**Responses still earn the point even if they...** • call `hailstoneLength` incorrectly.	**1 point** 3.C CON-1.E
		Total for part (b)	**2 points**

(c) `propLong`

	Scoring Criteria	Decision Rules	
6	Calls `isLongSeq` in the context of a loop		**1 point** 3.A MOD-1.G
7	Loops `1` to `n` (*no bounds errors*)		**1 point** 3.C CON-2.E
8	Calculates `double` proportion	**Responses still earn the point even if they...** • use incorrect values for the count of long sequences or `n`.	**1 point** 3.C CON-1.C
9	Returns correctly calculated value		**1 point** 3.B MOD-2.F
		Total for part (c)	**4 points**

Question-specific penalties

None

Total for question 1 **9 points**

Question 1: Calendar

Part (a)	`numberOfLeapYears`	**5 points**

Intent: *Return the number of leap years in a range*

+1 Initializes a numeric variable

+1 Loops through each necessary year in the range

+1 Calls `isLeapYear` on some valid year in the range

+1 Updates count based on result of calling `isLeapYear`

+1 Returns count of leap years

Part (b)	`dayOfWeek`	**4 points**

Intent: *Return an integer representing the day of the week for a given date*

+1 Calls `firstDayOfYear`

+1 Calls `dayOfYear`

+1 Calculates the value representing the day of the week

+1 Returns the calculated value

Question-Specific Penalties

-1 (t) Static methods called with `this`.

Question 1: Scoring Notes

Part (a) `numberOfLeapYears`			**5 points**
Points	Rubric Criteria	Responses earn the point even if they...	Responses will not earn the point if they...
+1	Initializes a numeric variable		• use the variable for loop control only
+1	Loops through each necessary year in the range		• consider years outside the range
+1	Calls `isLeapYear` on some valid year in the range	• do not use a loop	
+1	Updates count based on result of calling `isLeapYear`	• do not use a loop • do not initialize the counter	• use result as a non-`boolean`
+1	Returns count of leap years	• loop from `year1` to `year2` incorrectly • do not initialize the counter	• do not use a loop • update or initialize the counter incorrectly • return early inside the loop
Part (b) `dayOfWeek`			**4 points**
Points	Rubric Criteria	Responses earn the point even if they...	Responses will not earn the point if they...
+1	Calls `firstDayOfYear`		• do not use the given year
+1	Calls `dayOfYear`		• have arguments out of order
+1	Calculates the value representing the day of the week		• make any error in the calculation
+1	Returns the calculated value	• return the value from calling `firstDayOfYear` or `dayOfYear`	• return a constant value

3. 2019(2).1

Calendar

Part (a)

```
public static int numberOfLeapYears(int year1, int year2)
{
   int count = 0;
   for (int y = year1; y <= year2; y++)
   {
      if (isLeapYear(y))
      {
         count++;
      }
   }
   return count;
}
```

Part (b)

```
public static int dayOfWeek(int month, int day, int year)
{
   int startDay = firstDayOfYear(year);
   int nthDay = dayOfYear(month, day, year);
   int returnDay = (startDay + nthDay - 1) % 7;
   return returnDay;
}
```

Question 1: Frog Simulation

Part (a)	`simulate`	**5 points**

Intent: *Simulate the distance traveled by a hopping frog*

+1 Calls `hopDistance` and uses returned distance to adjust (or represent) the frog's position

+1 Initializes and accumulates the frog's position at most `maxHops` times (*must be in context of a loop*)

+1 Determines if a distance representing multiple hops is at least `goalDistance`

+1 Determines if a distance representing multiple hops is less than starting position

+1 Returns `true` if goal ever reached, `false` if goal never reached or position ever less than starting position

Part (b)	`runSimulations`	**4 points**

Intent: *Determine the proportion of successful frog hopping simulations*

+1 Calls `simulate` the specified number of times (*no bounds errors*)

+1 Initializes and accumulates a count of `true` results

+1 Calculates proportion of successful simulations using `double` arithmetic

+1 Returns calculated value

Question 1: Scoring Notes

Part (a) `simulate`			**5 points**
Points	Rubric Criteria	Responses earn the point if they...	Responses will not earn the point if they...
+1	Calls `hopDistance` and uses returned distance to adjust (or represent) the frog's position	• use `hopDistance()` as a position, like `hopDistance() < 0`	• only use `hopDistance()` as a count, like `hopDistance() < maxHops`
+1	Initializes and accumulates the frog's position at most `maxHops` times (*must be in context of a loop*)		• do not use a loop
+1	Determines if a distance representing multiple hops is at least `goalDistance`	• use some number of hops `*` `hopDistance()` as the frog's final position	
+1	Determines if a distance representing multiple hops is less than starting position		
+1	Returns `true` if goal ever reached, `false` if goal never reached or position ever less than starting position	• have checks for all three conditions and correct return logic based on those checks, even if a check did not earn a point	• do not check all three conditions • only check for `goalDistance` after the loop • only check for starting position after the loop
Part (b) `runSimulations`			**4 points**
Points	Rubric Criteria	Responses earn the point if they...	Responses will not earn the point if they...
+1	Calls `simulate` the specified number of times (*no bounds errors*)	• do not use the result of calling `simulate`	• do not use a loop
+1	Initializes and accumulates a count of `true` results		• initialize the count inside a loop • do not use a loop
+1	Calculates proportion of successful simulations using `double` arithmetic	• perform the correct calculation on an accumulated value, even if there was an error in the accumulation	• fail to divide by the parameter
+1	Returns calculated value		• calculate values using nonnumeric types • return a count of simulations

4. 2018.1

Frog Simulation

Part (a)

```
public boolean simulate()
{
   int position = 0;

   for (int count = 0; count < maxHops; count++)
   {
      position += hopDistance();
      if (position >= goalDistance)
      {
         return true;
      }
      else if (position < 0)
      {
         return false;
      }
   }
   return false;
}
```

Part (b)

```
public double runSimulations(int num)
{
   int countSuccess = 0;

   for (int count = 0; count < num; count++)
   {
      if(simulate())
      {
         countSuccess++;
      }
   }
   return (double)countSuccess / num;
}
```

Array FRQ Answer

1. 2005.4

PART A:

```
public double average(int first, int last)
{
  double sum = 0.0;
  for (int i = first; i <= last; i++)
  {
    sum += scores[i];
  }
    return sum/(last-first+1);
}
```

PART B:

```
public boolean hasImproved()
{
  for (int k = 0; k < scores.length-1; k++)
  {
    if (scores[k] > scores[k+1])
    {
      return false;
    }
  }
  return true;
}
```

PART C:

```
public double finalAverage()
{
  if (hasImproved())
  {
    return average(scores.length/2, scores.length-1);
  }
  else
  {
    return average(0, scores.length-1);
  }
}
```

2012.4

Question 4: GrayImage

Part (a):

```
public int countWhitePixels() {
   int whitePixelCount = 0;
   for (int[] row : this.pixelValues) {
      for (int pv : row) {
         if (pv == this.WHITE) {
            whitePixelCount++;
         }
      }
   }
   return whitePixelCount;
}
```

Part (a): Alternative solution

```
public int countWhitePixels() {
   int whitePixelCount = 0;
   for (int row = 0; row < pixelValues.length; row++) {
      for (int col = 0; col < pixelValues[0].length; col++) {
         if (pixelValues[row][col] == WHITE) {
            whitePixelCount++;
         }
      }
   }
   return whitePixelCount;
}
```

Part (b):

```
public void processImage() {
   for (int row = 0; row < this.pixelValues.length-2; row++) {
      for (int col = 0; col < this.pixelValues[0].length-2; col++) {
         this.pixelValues[row][col] -= this.pixelValues[row+2][col+2];
         if (this.pixelValues[row][col] < BLACK) {
            this.pixelValues[row][col] = BLACK;
         }
      }
   }
}
```

Part (b): Alternative solution

```
public void processImage() {
   for (int row = 0; row < this.pixelValues.length; row++) {
      for (int col = 0; col < this.pixelValues[0].length; col++) {
         if (row + 2 < pixelValues.length &&
                col + 2 < pixelValues[row].length) {
            this.pixelValues[row][col] -= this.pixelValues[row+2][col+2];
            if (this.pixelValues[row][col] < BLACK) {
               this.pixelValues[row][col] = BLACK;
            }
         }
      }
   }
}
```

Question 1: Diverse Array

Part (a)	`arraySum`	**2 points**

Intent: *Compute and return sum of elements in 1D array* `arr`, *passed as parameter*

+1 Accesses all elements of `arr`, (*no bounds errors on* `arr`)

+1 Initializes, computes, and returns sum of elements

Part (b)	`rowSums`	**4 points**

Intent: *Compute and return 1D array containing sums of each row in the 2D array* `arr2D`, *passed as parameter*

+1 Constructs correctly-sized 1D array of ints

+1 Accesses all rows in `arr2D` (*no bounds errors on* `arr2D`)

+1 Computes sum of row in `arr2D` using `arraySum` and assigns to element in 1D array

+1 Returns 1D array where kth element is computed sum of corresponding row in 2D array for all rows

Part (c)	`isDiverse`	**3 points**

Intent: *Determine whether* `arr2D`, *passed as parameter, is diverse*

+1 Computes and uses array of row sums from `arr2D` using `rowSums`

+1 Compare all and only pairs of row sums for equality (*No bounds errors on row sums array; point not awarded if no adjustment when compares any row sum with itself*)

+1 Returns `true` if all compared row sums are different and `false` otherwise (*point not awarded for immediate return*)

Question-Specific Penalties

-1 (g) Uses `getLength/getSize` for array size

-1 (y) Destruction of persistent data (`arr` *or* `arr2D`)

2015.1

Question 1: Diverse Array

Part (a):

```
public static int arraySum(int[] arr){
       int sum=0;
       for (int elem : arr){
            sum += elem;
       }
       return sum;
}
```

Part (b):

```
public static int[] rowSums(int[][] arr2D){
        int [] sums=new int[arr2D.length];
        int rowNum=0;
        for(int[] row : arr2D){
            sums[rowNum]=arraySum(row);
            rowNum++;
        }
        return sums;
    }
```

Part (c):

```
public static boolean isDiverse(int[][] arr2D){
        int [] sums=rowSums(arr2D);
        for (int i=0; i < sums.length; i++){
            for (int j=i+1; j < sums.length; j++){
                if (sums[i]==sums[j]){
                    return false;
                }
            }
        }
        return true;
    }
```

Question 4: Latin Squares

Part (a) `getColumn` **4 points**

Intent: *Create a 1-D array that contains the values from one column of a 2-D array*

+1 Constructs a new `int` array of size `arr2D.length`

+1 Accesses all items in one column of `arr2D` (*no bounds errors*)

+1 Assigns one element from `arr2D` to the corresponding element in the new array

+1 **On exit:** The new array has all the elements from the specified column in `arr2D` in the correct order

Part (b) `isLatin` **5 points**

Intent: *Check conditions to determine if a square 2-D array is a Latin square*

+1 Calls `containsDuplicates` referencing a row or column of `square`

+1 Calls `hasAllValues` referencing two different rows, two different columns, or one row and one column

+1 Applies `hasAllValues` to all rows or all columns (*no bounds errors*)

+1 Calls `getColumn` to obtain a valid column from `square`

+1 Returns `true` if all three Latin square conditions are satisfied, `false` otherwise

Question-Specific Penalties

-1 (r) incorrect construction of a copy of a row

-1 (s) syntactically incorrect method call to any of `getColumn()`, `containsDuplicates()`, or `hasAllValues()`

Question 4: Scoring Notes

Part (a) `getColumn`			**4 points**
Points	Rubric Criteria	Responses earn the point if they…	Responses will not earn the point if they…
+1	Constructs a new `int` array of size `arr2D.length`		• only create an `ArrayList`
+1	Accesses all items in one column of `arr2D` (*no bounds errors*)	• declare the new array of an incorrect size and use that size as the number of loop iterations	• switch row and column indices
+1	Assigns one element from `arr2D` to the corresponding element in the new array		• use `ArrayList` methods to add to array
+1	**On exit:** The new array has all the elements from the specified column in `arr2D` in the correct order		• switch row and column indices • do not use an index when assigning values to the array
Part (b) `isLatin`			**5 points**
Points	Rubric Criteria	Responses earn the point if they…	Responses will not earn the point if they…
+1	Calls `containsDuplicates` referencing a row or column of `square`	• reference any row or column of `square`, even if the syntax of the reference is incorrect	
+1	Calls `hasAllValues` referencing two different rows, two different columns, or one row and one column	• reference any two distinct rows, two distinct columns, or a row and column of `square`, even if the syntax of the reference is incorrect	
+1	Applies `hasAllValues` to all rows or all columns (*no bounds errors*)		• only reference one array in the call to `hasAllValues`
+1	Calls `getColumn` to obtain a valid column from `square`		• reverse parameters
+1	Returns `true` if all three Latin square conditions are satisfied, `false` otherwise	• test the three sets of conditions and return the correct value	

Return is not assessed in Part (a).

2018.4

Latin Squares

Part (a)

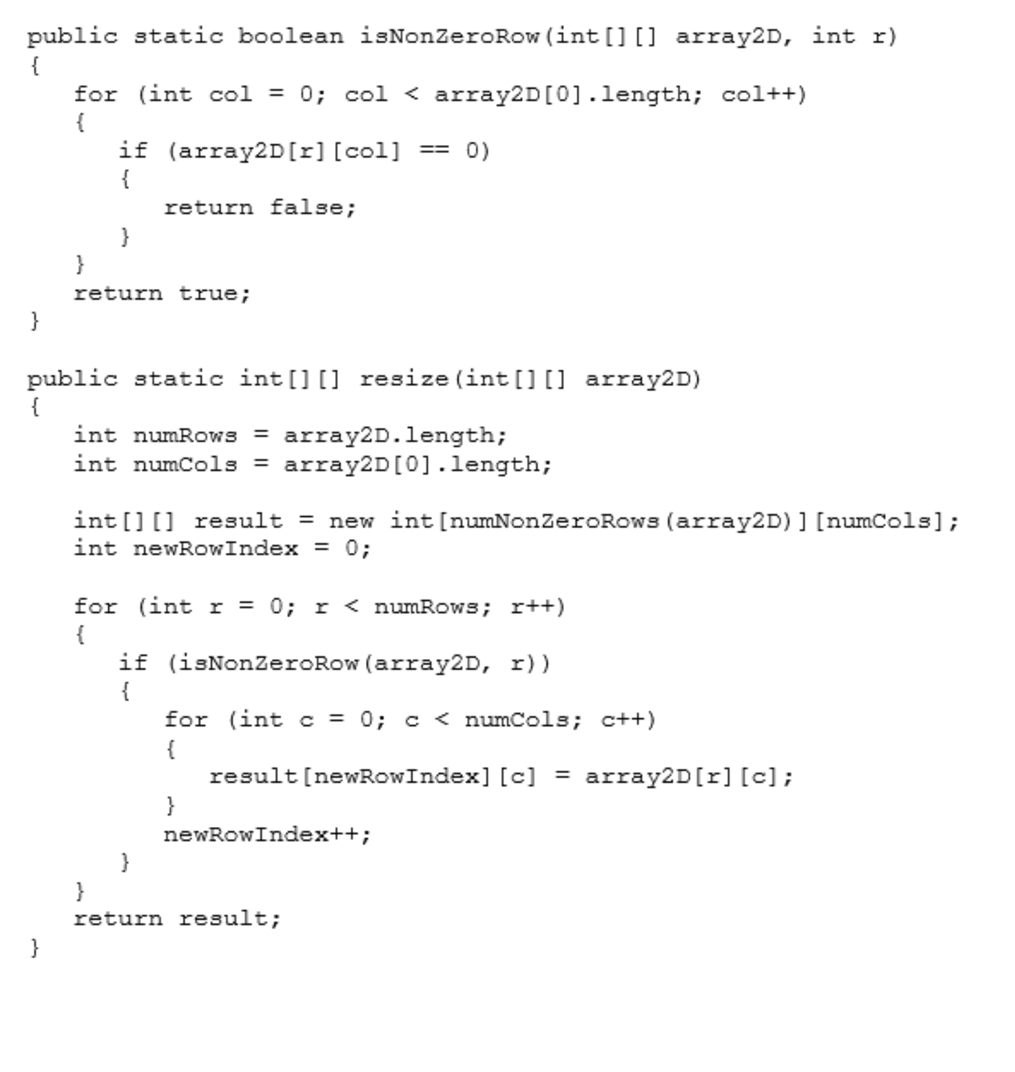
```
public static int[] getColumn(int[][] arr2D, int c)
{
   int[] result = new int[arr2D.length];

   for (int r = 0; r < arr2D.length; r++)
   {
      result[r] = arr2D[r][c];
   }
   return result;
}
```

Part (b)

```
public static boolean isLatin(int[][] square)
{
   if (containsDuplicates(square[0]))
   {
      return false;
   }

   for (int r = 1; r < square.length; r++)
   {
      if (!hasAllValues(square[0], square[r]))
      {
         return false;
      }
   }

   for (int c = 0; c < square[0].length; c++)
   {
      if (!hasAllValues(square[0], getColumn(square, c)))
      {
         return false;
      }
   }

   return true;
}
```

Question 4: 2D Array 9 points

Canonical solution



(a) **3 points**

```
public static boolean isNonZeroRow(int[][] array2D, int r)
{
   for (int col = 0; col < array2D[0].length; col++)
   {
      if (array2D[r][col] == 0)
      {
         return false;
      }
   }
   return true;
}
```

(b) **6 points**

```
public static int[][] resize(int[][] array2D)
{
   int numRows = array2D.length;
   int numCols = array2D[0].length;

   int[][] result = new int[numNonZeroRows(array2D)][numCols];
   int newRowIndex = 0;

   for (int r = 0; r < numRows; r++)
   {
      if (isNonZeroRow(array2D, r))
      {
         for (int c = 0; c < numCols; c++)
         {
            result[newRowIndex][c] = array2D[r][c];
         }
         newRowIndex++;
      }
   }
   return result;
}
```

(a) `isNonZero`

	Scoring Criteria	Decision Rules	
1	Compares an item from `array2D` with `0`	Responses **will not** earn the point if they fail to attempt the comparison, even if they access an item from `array2D`	**1 point**
2	Accesses every item from row `r` of 2D array (*no bounds errors*)	Responses **can** still earn the point even if they return early from an otherwise correctly-bounded loop	**1 point**
3	Returns `true` if and only if row contains no zeros	Responses **can** still earn the point even if they process a column of the 2D array rather than a row Responses **will not** earn the point if they fail to return a value in some cases	**1 point**
		Total for part (a)	**3 points**

(b) `resize`

	Scoring Criteria	Decision Rules	
4	Calls `numNonZeroRows` and `isNonZeroRow`	Responses **can** still earn the point even if they fail to use or store the return value Responses **will not** earn the point if they • include incorrect number or type of parameters • call methods on an object or class other than `ArrayResizer`	**1 point**
5	Identifies rows with no zeros (*in the context of an* `if`)	Responses **can** still earn the point even if they call `isNonZeroRow` incorrectly, if the row being tested is clearly identified (index or reference)	**1 point**
6	Declares and creates a new 2D array of the correct size	Response **will not** earn the point if they transpose the dimensions of the created array	**1 point**
7	Maintains an index in the new array	Responses **will not** earn the point if they • fail to declare, initialize, and update a different index • maintain the index in a way that overwrites, skips, or duplicates rows	**1 point**
8	Traverses all necessary elements of `array2D` (*no bounds errors*)	Responses **can** still earn the point even if they • cause a bounds error by declaring and creating a new 2D array of an incorrect size • fail to maintain an index in the new array correctly, resulting in a bounds error • fail to access individual elements in a nested loop, if they access each row as an entire row Responses **will not** earn the point if they transpose coordinates, leading to a bounds error and/or copying columns	**1 point**
9	Copies all and only rows identified as having no zero elements into the new array	Responses **can** still earn the point even if they • copy a reference • identify rows incorrectly, if the logical sense can be determined and is correct • copy columns instead of rows, consistent with the dimensions of the created 2D array	**1 point**

	Responses **will not** earn the point if they • remove or overwrite data from `array2D` (instead of or in addition to copying it to the new array) • reverse the logical sense of which rows to copy	
	Total for part (b)	**6 points**

Question-specific penalties

-1 (u) Use `array2D[].length` to refer to the number of columns in a row of the 2D array

Total for question 4 **9 points**

2013.4

Question 4: SkyView

Part (a):

```
public SkyView(int numRows, int numCols, double[] scanned)
{
  view = new double[numRows][numCols];
  int i = 0;
  for (int row = 0; row < numRows; row++) {
    if (row % 2 == 0) {
      for (int col = 0; col < numCols; col++) {
        view[row][col] = scanned[i];
        i++;
      }
    }
    else {
      for (int col = numCols - 1; col >= 0; col--) {
        view[row][col] = scanned[i];
        i++;
      }
    }
  }
}
```

Part (b):

```
public double getAverage(int startRow, int endRow, int startCol,
                                                   int endCol)
{
    double sum = 0.0;
    int count = 0;
    for (int row = startRow; row <= endRow; row++){
        for (int col = startCol; col <= endCol; col++){
            sum += view[row][col];
            count++;
        }
    }
    return sum / count;
}
```

Question 4: Light Board

Part (a) `LightBoard` **4 points**

Intent: *Define implementation of a constructor that initializes a 2D array of lights*

+1 Creates a `new boolean[numRows][numCols]` and assigns to instance variable `lights`

+1 Accesses all elements in the created 2D array (*no bounds errors*)

+1 Computes the 40% probability

+1 Sets all values of 2D array based on computed probability

Part (b) `evaluateLight` **5 points**

Intent: *Evaluate the status of a light in a 2D array of lights*

+1 Accesses an element of `lights` as a `boolean` value in an expression

+1 Traverses specified `col` of a 2D array (*no bounds errors*)

+1 Counts the number of `true` values in the traversal

+1 Performs an even calculation and a multiple of three calculation

+1 Returns `true` or `false` according to all three rules

Question-Specific Penalties

-1 (z) Constructor returns a value

-1 (y) Destruction of persistent data

Question 4: Scoring Notes

Part (a) `LightBoard`			**4 points**
Points	Rubric Criteria	Responses earn the point even if they…	Responses will not earn the point if they…
+1	Creates a `new boolean[numRows][numCols]` and assigns to instance variable `lights`		• initialize a local variable that is never assigned to `lights` • omit the keyword `new` • use a type other than `boolean`
+1	Accesses all elements in the created 2D array (*no bounds errors*)	• fail to create `lights` but assume `lights[numRows][numCols]`	
+1	Computes the 40% probability	• use `Math.random() <= .4`	• incorrectly cast to `int`
+1	Sets all values of 2D array based on computed probability	• only assign `true` values	• compute a single probability but use it multiple times • reverse the sense of the comparison when assigning
Part (b) `evaluateLight`			**5 points**
Points	Rubric Criteria	Responses earn the point even if they…	Responses will not earn the point if they…
+1	Accesses an element of `lights` as a `boolean` value in an expression		• access `lights` as a type other than `boolean`
+1	Traverses specified `col` of a 2D array (*no bounds errors*)		
+1	Counts the number of `true` values in the traversal	• access too many or too few items in a single column • access a single row instead of a single column	• count an item more than once
+1	Performs an even calculation and a multiple of three calculation		• use `/` instead of `%`
+1	Returns `true` or `false` according to all three rules	• have an incorrect column count but use the correct logic	• fail to return a value in some case • implement counting loop more than once with one loop that is incorrect

2019(2).4

Light Board

Part (a)

```
public LightBoard(int numRows, int numCols)
{
   lights = new boolean[numRows][numCols];

   for (int r = 0; r < numRows; r++)
   {
      for (int c = 0; c < numCols; c++)
      {
         double rnd = Math.random();
         lights[r][c] = rnd < 0.4;
      }
   }
}
```

Part (b)

```
public boolean evaluateLight(int row, int col)
{
   int numOn = 0;

   for (int r = 0; r < lights.length; r++)
   {
      if (lights[r][col])
      {
         numOn++;
      }
   }

   if (lights[row][col] && numOn % 2 == 0)
   {
      return false;
   }
   if (!lights[row][col] && numOn % 3 == 0)
   {
      return true;
   }
   return lights[row][col];
}
```

Question 4: Successor Array

Part (a) `findPosition` **5 points**

Intent: *Find the position of a given integer in a 2D integer array*

+1 Accesses all necessary elements of `intArr` (*no bounds errors*)

+1 Identifies `intArr` element equal to `num` (*in context of an* `intArr` *traversal*)

+1 Constructs `Position` object with same row and column as identified `intArr` element

+1 Selects constructed object when `intArr` element identified; `null` when not

+1 Returns selected value

Part (b) `getSuccessorArray` **4 points**

Intent: *Create a successor array based on a 2D integer array*

+1 Creates 2D array of `Position` objects with same dimensions as `intArr`

+1 Assigns a value to a location in 2D successor array using a valid call to `findPosition`

+1 Determines the successor `Position` of an `intArr` element accessed by row and column (*in context of* `intArr` *traversal*)

+1 Assigns all necessary locations in successor array with corresponding position object or `null` (*no bounds errors*)

Question-Specific Penalties

-1 (s) Uses confused identifier `Arr`

-1 (t) Uses `intArr[].length` as the number of columns

-1 (u) Uses non-existent accessor methods from `Position`

Question 4: Scoring Notes

Part (a) `findPosition`			**5 points**
Points	Rubric Criteria	Responses earn the point if they ...	Responses will not earn the point if they ...
+1	Accesses all necessary elements of `intArr` (*no bounds errors*)		• use `if (...) return; else return null;` inside loop • confuse row and column bounds • fail to traverse `intArr`
+1	Identifies `intArr` element equal to `num` (*in context of an `intArr` traversal*)		• use `.equals` instead of `==`
+1	Constructs `Position` object with same row and column as identified `intArr` element		• omit keyword `new` • use `(r,c)` instead of `Position(r,c)`
+1	Selects constructed object when `intArr` element identified; `null` when not	• use `"null"` instead of `null` • construct a `String` object using row and column indices	• use `if (...) return; else return null;` inside loop • use `(r,c)` instead of `Position(r,c)`
+1	Returns selected value		
Part (b) `getSuccessorArray`			**4 points**
Points	Rubric Criteria	Responses earn the point if they ...	Responses will not earn the point if they ...
+1	Creates 2D array of `Position` objects with same dimensions as `intArr`		• omit keyword `new`
+1	Assigns a value to a location in 2D successor array using a valid call to `findPosition`	• call `Successors.findPosition(…)`	• reimplement the code from `findPosition` • call `findPosition` with a single argument • call `this.findPosition(…)`
+1	Determines the successor `Position` of an `intArr` element accessed by row and column (*in context of `intArr` traversal*)	• reimplement the code from `findPosition`	• call `findPosition` using an integer that is not identified with a location in `intArr` • call `findPosition` with a single argument
+1	Assigns all necessary locations in successor array with corresponding position object or `null` (*no bounds errors*)	• use `SuccessorArray` dimensions correctly, even if `SuccessorArray` was not initialized properly • only assign non-null entries to `SuccessorArray`	• reimplement the code from `findPosition` but mishandle the null case. • fail to traverse `intArr`

Return is not assessed in Part (b).

2017.4

Successor Array

Part (a)

```
public static Position findPosition(int num, int[][] intArr)
{
      for (int row=0; row < intArr.length; row++)
      {
            for (int col=0; col < intArr[0].length; col++)
            {
                  if (intArr[row][col] == num)
                  {
                        return new Position(row, col);
                  }
            }
      }
      return null;
}
```

Part (b)

```
public static Position[][] getSuccessorArray(int[][] intArr)
{
      Position[][] newArr = new Position[intArr.length][intArr[0].length];

      for (int row=0; row < intArr.length; row++)
      {
            for (int col=0; col < intArr[0].length; col++)
            {
                  newArr[row][col] = findPosition(intArr[row][col]+1, intArr);
            }
      }
      return newArr;
}
```

2019(1).4

Question 4: 2D Array — 9 points

Learning Objectives: MOD-1.D.b MOD-1.G CON-1.H CON-2.A CON-2.N.c VAR-2.F VAR-2.G.a

Canonical solution

(a) **5 points**

```
public Theater(int seatsPerRow, int tier1Rows,
               int tier2Rows)
{
   theaterSeats =
      new Seat[tier1Rows + tier2Rows][seatsPerRow];
   for (int r = 0; r < tier1Rows + tier2Rows; r++)
   {
      for (int c = 0; c < seatsPerRow; c++)
      {
         if (r < tier1Rows)
         {
            theaterSeats[r][c] = new Seat(true, 1);
         }
         else
         {
            theaterSeats[r][c] = new Seat(true, 2);
         }
      }
   }
}
```

(b) **4 points**

```
public boolean reassignSeat(int fromRow, int fromCol,
                            int toRow, int toCol)
{
   Seat toS = theaterSeats[toRow][toCol];
   if (!toS.isAvailable())
   {
      return false;
   }

   Seat fromS = theaterSeats[fromRow][fromCol];
   if (toS.getTier() < fromS.getTier())
   {
      return false;
   }

   toS.setAvailability(false);
   fromS.setAvailability(true);
   return true;
}
```

(a) **`Theater`**

	Scoring Criteria	Decision Rules	
1	Instantiates a new `Seat[][]` with the correct number of rows and columns, based on parameters		**1 point** 3.E VAR-2.F
2	Traverses the `theaterSeats` array (*no bounds errors*)		**1 point** 3.E VAR-2.G.a
3	Instantiates a new `Seat` object with a tier and availability status	**Responses still earn the point even if they...** • incorrectly assign the new object to a `theaterSeats` element.	**1 point** 3.A MOD-1.D.b
4	Accesses a `theaterSeats` element and assigns it a new `Seat` object	**Responses still earn the point even if they...** • incorrectly instantiate the new `Seat` object; or • assign the new `Seat` object to an incorrect `theaterSeats` element.	**1 point** 3.E VAR-2.F
5	Correct tiers assigned to all array elements		**1 point** 3.C CON-2.A
		Total for part (a)	**5 points**

(b) `reassignSeat`

	Scoring Criteria	Decision Rules	
6	Accesses *from* and *to* `Seat` objects		**1 point** 3.E VAR-2.F
7	Calls `isAvailable` and `getTier` on `Seat` objects	**Responses still earn the point even if they...** • correctly call methods on `theaterSeats` elements other than the *to* and *from* seats.	**1 point** 3.A MOD-1.G
8	Checks if move can be made based on both tiers and the availability status of *to* `Seat` object		**1 point** 3.C CON-1.H
9	Correctly updates availability of both seats and returns `true` if the move can be made; otherwise, returns `false`		**1 point** 3.E CON-2.N.c
		Total for part (b)	**4 points**

Question-specific penalties

None

		Total for question 4	**9 points**

Question 3: Crossword

Part (a)	`toBeLabeled`	**3 points**

Intent: *Return a* `boolean` *value indicating whether a crossword grid square should be labeled with a positive number*

+1 Checks `blackSquares[r][c]`

+1 Checks for black square/border above and black square/border to the left (*no bounds errors*)

+1 Returns `true` if square should be labeled with positive number; returns `false` otherwise

Part (b)	`Crossword` constructor	**6 points**

Intent: *Initialize each square in a crossword puzzle grid to have a color* `(boolean)` *and an integer label*

+1 `puzzle = new Square[blackSquares.length][blackSquares[0].length];` (*or equivalent*)

+1 Accesses all locations in `puzzle` (*no bounds errors*)

+1 Calls `toBeLabeled` with appropriate parameters

+1 Creates and assigns new `Square` to location in `puzzle`

+1 Numbers identified squares consecutively, in row-major order, starting at 1

+1 On exit: All squares in `puzzle` have correct color and number (*minor errors covered in previous points ok*)

Question-Specific Penalties

-2 (p) Consistently uses incorrect name instead of `puzzle`

-1 (q) Uses *array*`[].length` instead of *array*`[`*num*`].length`

2016.3

Crossword

Part (a):

```
private boolean toBeLabeled(int r, int c, boolean[][] blackSquares)
{
     return (!(blackSquares[r][c]) &&
            (r == 0 || c == 0 || blackSquares[r - 1][c] ||
             blackSquares[r][c - 1]));
}
```

Part (b):

```
public Crossword(boolean[][] blackSquares)
{
     puzzle = new Square[blackSquares.length][blackSquares[0].length];
     int num = 1;

     for (int r = 0; r < blackSquares.length; r++)
     {
          for (int c = 0; c < blackSquares[0].length; c++)
          {
               if (blackSquares[r][c])
               {
                    puzzle[r][c] = new Square(true, 0);
               }
               else
               {
                    if (toBeLabeled(r, c, blackSquares))
                    {
                         puzzle[r][c] = new Square(false, num);
                         num++;
                    }
                    else
                    {
                         puzzle[r][c] = new Square(false, 0);
                    }
               }
          }
     }
}
```

2009.3

Battery Charger

PART A:

```
/** Determines the total cost to charge the battery starting
 *    at the beginning of startHour.
 *  @param startHour the hour at which the charge period begins
 *         Precondition: 0 ≤ startHour ≤ 23
 *  @param chargeTime the number of hours the battery needs to be charged
 *         Precondition: chargeTime > 0
 *  @return the total cost to charge the battery
 */
private int getChargingCost(int startHour, int chargeTime)
{
  int cost = 0;
  for (int x = 0; x < chargeTime; x++)
  {
    cost += this.rateTable[(startHour + x) % 24];
  }
  return cost;
}
```

PART B:

```
/** Determines start time to charge the battery at the lowest
 *    cost for the given charge time.
 *  @param chargeTime the number of hours the battery needs to be charged
 *         Precondition: chargeTime > 0
 *  @return an optimal start time, with 0 ≤ returned value ≤ 23
 */
public int getChargeStartTime(int chargeTime)
{
  int startTime = 0;
  for (int i = 1; i < 24; i++)
  {
    if (this.getChargingCost(i, chargeTime)
        < this.getChargingCost(startTime, chargeTime))
    {
      startTime = i;
    }
  }
  return startTime;
}
```

2014.3

SeatingChart

Part (a):

```
public SeatingChart(List<Student> studentList, int rows, int cols){
   seats=new Student[rows][cols];
   int studentIndex=0;
   for (int col = 0; col < cols; col++){
      for (int row = 0; row < rows; row++){
         if (studentIndex < studentList.size()){
            seats[row][col] = studentList.get(studentIndex);
            studentIndex++;
         }
      }
   }
}
```

Part (a) alternate:

```
public SeatingChart(List<Student> studentList, int rows, int cols){
   seats=new Student[rows][cols];
   int row=0;
   int col=0;
   for (Student student : studentList){
      seats[row][col]=student;
      row++;
      if (row==rows){
         row=0;
         col++;
      }
   }
}
```

Part (b):

```
public int removeAbsentStudents(int allowedAbsences){
   int count = 0;
   for (int row=0; row < seats.length; row++){
      for (int col=0; col < seats[0].length; col++){
         if (seats[row][col] != null &&
               seats[row][col].getAbsenceCount() > allowedAbsences){
            seats[row][col]=null;
            count++;
         }
      }
   }
   return count;
}
```

2013.2

TokenPass

Part (a):

```
public TokenPass(int playerCount)
{
    board = new int[playerCount];
    for (int i = 0; i < playerCount; i++){
        board[i] = 1 + (int) (10 * Math.random());
    }
    currentPlayer = (int) (playerCount * Math.random());
}
```

Part (b):

```
public void distributeCurrentPlayerTokens()
{
    int nextPlayer = currentPlayer;
    int numToDistribute = board[currentPlayer];
    board[currentPlayer] = 0;

    while (numToDistribute > 0){
        nextPlayer = (nextPlayer + 1) % board.length;
        board[nextPlayer]++;
        numToDistribute--;
    }
}
```

2012.3

Question 3: Horse Barn

Part (a):

```
public int findHorseSpace(String name) {
   for (int i = 0; i < this.spaces.length; i++) {
      if (this.spaces[i]!=null && name.equals(this.spaces[i].getName())) {
         return i;
      }
   }
   return -1;
}
```

Part (b):

```
public void consolidate() {
   for (int i = 0; i < this.spaces.length-1; i++) {
      if (this.spaces[i] == null) {
         for (int j = i+1; j < this.spaces.length; j++) {
            if (this.spaces[j] != null) {
               this.spaces[i] = this.spaces[j];
               this.spaces[j] = null;
               j = this.spaces.length;
            }
         }
      }
   }
}
```

Part (b): Alternative solution (auxiliary with array)

```
public void consolidate() {
   Horse[] newSpaces = new Horse[this.spaces.length];
   int nextSpot = 0;
   for (Horse nextHorse : this.spaces) {
      if (nextHorse != null) {
         newSpaces[nextSpot] = nextHorse;
         nextSpot++;
      }
   }
   this.spaces = newSpaces;
}
```

Part (b): Alternative solution (auxiliary with `ArrayList`)

```
public void consolidate() {
   List<Horse> horseList = new ArrayList<Horse>();
   for (Horse h : this.spaces) {
      if (h != null) horseList.add(h);
   }
   for (int i = 0; i < this.spaces.length; i++) {
      this.spaces[i] = null;
   }
   for (int i = 0; i < horseList.size(); i++) {
      this.spaces[i] = horseList.get(i);
   }
}
```

2007.1

1: Self Divisor

PART A:

```
public static boolean isSelfDivisor(int number) {
     int n = number;
     while (n > 0) {
         int digit = n % 10;
         if (digit == 0 || number % digit != 0) {
             return false;
         }
         n /= 10;
     }
     return true;
}
```

ALTERNATE SOLUTION:

```
public static boolean isSelfDivisor(int number) {
     String str = "" + number;
     for (int i = 0; i < str.length(); i++) {
         int digit = Integer.parseInt(str.substring(i,i+1));
         if (digit == 0 || number % digit != 0) {
             return false;
         }
     }
     return true;
}
```

PART B:

```
public static int[] firstNumSelfDivisors(int start, int num) {
    int[] selfs = new int[num];
    int numStored = 0;
    int nextNumber = start;
    while (numStored < num) {
        if (isSelfDivisor(nextNumber)) {
            selfs[numStored] = nextNumber;
            numStored++;
        }
        nextNumber++;
    }
    return selfs;
}
```

ALTERNATE SOLUTION:

```
public static int[] firstNumSelfDivisors(int start, int num) {
    int[] selfs = new int[num];
    int numStored = 0;
    int nextNumber = start;
    for (int i = 0; i < num; i++) {
        while (!isSelfDivisor(nextNumber)) {
            nextNumber++;
        }
        selfs[numStored] = nextNumber;
        numStored++;
        nextNumber++;
    }
    return selfs;
}
```

2011.1

Question 1: Sound

Part (a):

```
public int limitAmplitude(int limit) {
  int numChanged = 0;
  for (int i = 0; i < this.samples.length; i++) {
    if (this.samples[i] < -limit) {
      this.samples[i] = -limit;
      numChanged++;
    }
    if (this.samples[i] > limit) {
      this.samples[i] = limit;
      numChanged++;
    }
  }
  return numChanged;
}
```

Part (b):

```
public void trimSilenceFromBeginning() {
  int i = 0;
  while (this.samples[i] == 0) {
    i++;
  }
  int[] newSamples = new int[this.samples.length - i];
  for (int j = 0; j < newSamples.length; j++) {
    newSamples[j] = this.samples[j+i];
  }
  this.samples = newSamples;
}
```

2010.3

Question 3: Trail

Part (a):

```
public boolean isLevelTrailSegment(int start, int end) {
  int min = this.markers[start];
  int max = this.markers[start];
  for (int i = start + 1; i <= end; i++) {
    if (min > this.markers[i]) {
      min = this.markers[i];
    }
    if (max < this.markers[i]) {
      max = this.markers[i];
    }
  }
  return ((max - min) <= 10);
}

// Alternative solution (compares differences; uses early return):

public boolean isLevelTrailSegment(int start, int end) {
  for (int i = start; i < end; i++) {
    for (int j = start + 1; j <= end; j++) {
      if (Math.abs(this.markers[i] - this.markers[j]) > 10) {
        return false;
      }
    }
  }
  return true;
}
```

Part (b):

```
public boolean isDifficult() {
  int numChanges = 0;
  for (int i = 0; i < this.markers.length - 1; i++) {
    if (Math.abs(this.markers[i] - this.markers[i + 1]) >= 30) {
      numChanges++;
    }
  }
  return (numChanges >= 3);
}
```

2009.1

Question 1: Number Cube

PART A:

```
/** Returns an array of the values obtained by tossing
 *    a number cube numTosses times.
 *  @param cube a NumberCube
 *  @param numTosses the number of tosses to be recorded
 *         Precondition: numTosses > 0
 *  @return an array of numTosses values
 */
public static int[] getCubeTosses(NumberCube cube, int numTosses)
{
  int[] cubeTosses = new int[numTosses];
  for (int i = 0; i < numTosses; i++)
  {
    cubeTosses[i] = cube.toss();
  }
  return cubeTosses;
}
```

Question 1: Number Cube (continued)

PART B:

```
/** Returns the starting index of a longest run of two or more
 *    consecutive repeated values in the array values.
 *  @param values an array of integer values representing a series
 *    of number cube tosses
 *    Precondition: values.length > 0
 *  @return the starting index of a run of maximum size;
 *          -1 if there is no run
 */
public static int getLongestRun(int[] values)
{
  int currentLen = 0;
  int maxLen = 0;
  int maxStart = -1;
  for (int i = 0; i < values.length-1; i++)
  {
    if (values[i] == values[i+1])
    {
      currentLen++;
      if (currentLen > maxLen)
      {
        maxLen = currentLen;
        maxStart = i - currentLen + 1;
      }
    }
    else
    {
      currentLen = 0;
    }
  }
  return maxStart;
}
```

```
public static int getLongestRun(int[] values) {
        int index = -1;
        int maxLength = 1;
        int currentLength = 1;
        for (int i = 0; i < values.length - 1; i++) {
            if (values[i] == values[i + 1])
                currentLength++;
            else
                currentLength = 1;

            if (currentLength > maxLength) {
                index = i - currentLength + 2;
                maxLength = currentLength;
            }
        }
        return index;
}
```

```
public static int getLongestRun(int[] values) {
    int maxLength = 1;
    int startIndex = -1;
    int currentLength = 1;

    for (int i = values.length - 2; i >= 0; i--) {
        if (values[i] == values[i + 1]) {
            currentLength++;
            if (currentLength > maxLength)
             {
                 maxLength = currentLength;
                 startIndex = i;
            }
        } else
            currentLength = 1;
        }

    return startIndex;
}
```

Question 1: Number Cube (continued)

PART B (ALTERNATE SOLUTION I):

```
public static int getLongestRun(int[] values)
{
  int maxStart = -1;
  int maxLen = -1;
  int currentLen = 0;
  int currVal = -1;
  int currStart = 0;
  for (int i = 0; i < values.length; i++)
  {
    if (values[i] == currVal)
      currentLen++;
    else
    {
      if (currentLen > maxLen)
      {
        maxLen = currentLen;
        maxStart = currStart;
      }
      currStart = i;
      currentLen = 1;
      currVal=values[i];
    }
  }

  if (currentLen > maxLen)
  {
    maxLen = currentLen;
    maxStart = currStart;
  }
  if (maxLen == 1)
    return -1;
  else
    return maxStart;
}
```

Question 1: Number Cube (continued)

PART B (ALTERNATE SOLUTION II):

```
public static int getLongestRun(int[] values)
{
  int maxLen = 0;
  int currLen = 0;
  int index = -1;
  int currVal = -1;
  for (int i = values.length - 1; i >= 0; i--)
  {
    if (values[i] == currVal)
      currLen++;
    else
    {
      if (maxLen < currLen)
      {
        maxLen = currLen;
        index = i+1;
      }
      currVal = values[i];
      currLen = 1;
    }
  }

  if (maxLen < currLen)
  {
    maxLen = currLen;
    index = 0;
  }
  if (maxLen == 1)
    return -1;

  return index;
}
```

2006.3

Question 3: Customer List

PART A:

```
public int compareCustomer(Customer other)
{
  int nameCompare = getName().compareTo(other.getName());
  if (nameCompare != 0)
  {
    return nameCompare;
  }
  else
  {
    return getID() - other.getID();
  }
}
```

PART B:

```
public static void prefixMerge(Customer[] list1, Customer[] list2, Customer[] result)
{
  int front1 = 0;
  int front2 = 0;

  for (int i = 0; i < result.length; i++)
  {
    int comparison = list1[front1].compareCustomer(list2[front2]);
    if (comparison < 0)
    {
      result[i] = list1[front1];
      front1++;
    }
    else if (comparison > 0)
    {
      result[i] = list2[front2];
      front2++;
    }
    else
    {
      result[i] = list1[front1];
      front1++;
      front2++;
    }
  }
}
```

2005.1

PART A:

```
public Reservation requestRoom(String guestName)
{
  for (int i = 0; i < rooms.length; i++)
  {
    if (rooms[i] == null)
    {
      rooms[i] = new Reservation(guestName, i);
      return rooms[i];
    }
  }
  waitList.add(guestName);
  return null;
}
```

PART B:

```
public Reservation cancelAndReassign(Reservation res)
{
  int roomNum = res.getRoomNumber();
  if (waitList.isEmpty())
  {
    rooms[roomNum] = null;
  }
  else
  {
    rooms[roomNum] = new Reservation((String)waitList.get(0), roomNum)
    waitlist.remove(0);
  }
  return rooms[roomNum];
}
```

alternate solution

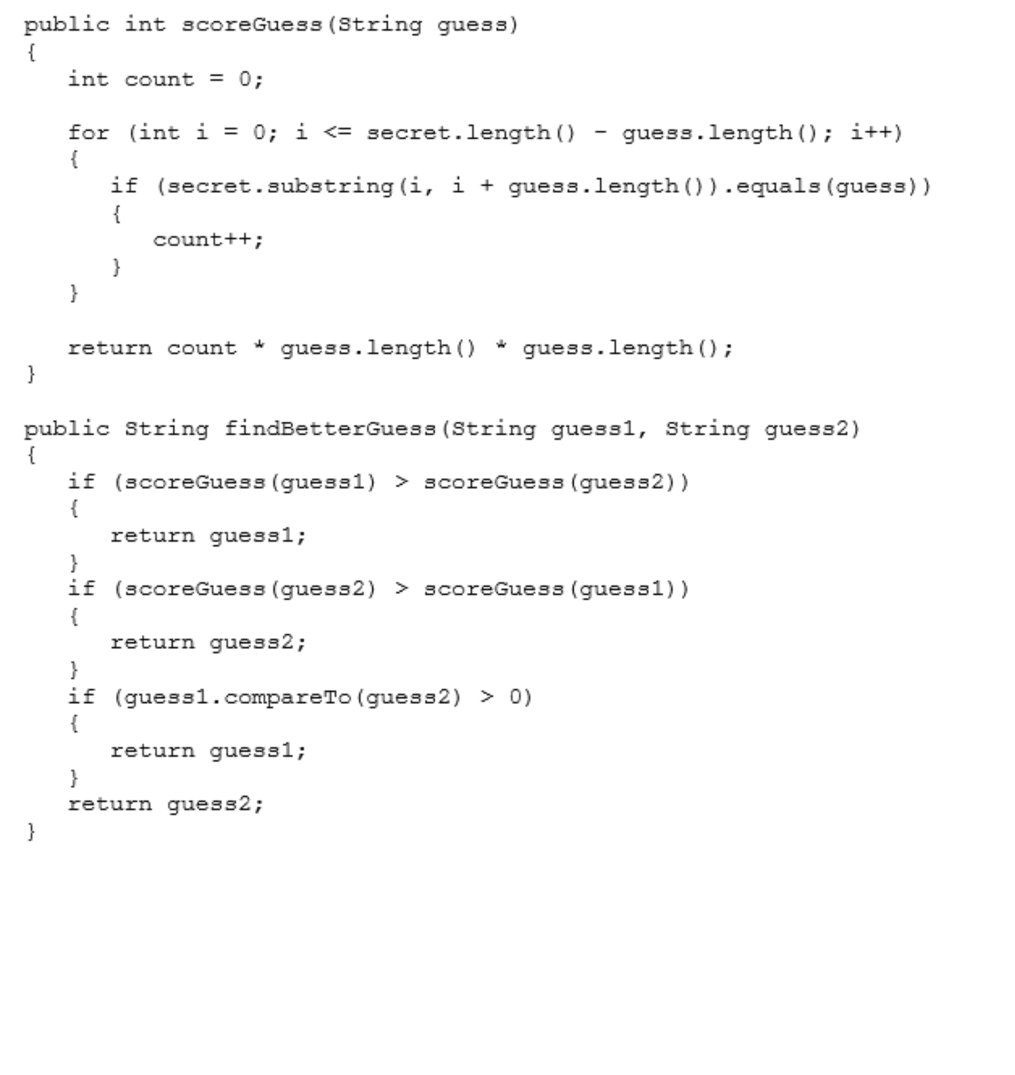
```
public Reservation cancelAndReassign(Reservation res)
{
  int roomNum = res.getRoomNumber();
  rooms[roomNum] = null;
  if (!waitList.isEmpty())
  {
    requestRoom((String)waitlist.get(0));
    waitlist.remove(0);
  }
  return rooms[roomNum];
}
```

String FRQ Answer

2021.1

Question 1: Methods and Control Structures — 9 points

Canonical solution



(a) — **5 points**

```
public int scoreGuess(String guess)
{
   int count = 0;

   for (int i = 0; i <= secret.length() - guess.length(); i++)
   {
      if (secret.substring(i, i + guess.length()).equals(guess))
      {
         count++;
      }
   }

   return count * guess.length() * guess.length();
}
```

(b) — **4 points**

```
public String findBetterGuess(String guess1, String guess2)
{
   if (scoreGuess(guess1) > scoreGuess(guess2))
   {
      return guess1;
   }
   if (scoreGuess(guess2) > scoreGuess(guess1))
   {
      return guess2;
   }
   if (guess1.compareTo(guess2) > 0)
   {
      return guess1;
   }
   return guess2;
}
```

(a) `scoreGuess`

	Scoring Criteria	Decision Rules	
1	Compares `guess` to a substring of `secret`	Responses **can** still earn the point even if they only call `secret.indexOf(guess)` Responses **will not** earn the point if they use `==` instead of `equals`	**1 point**
2	Uses a substring of `secret` with correct length for comparison with `guess`	Responses **can** still earn the point even if they • only call `secret.indexOf(guess)` • use `==` instead of `equals`	**1 point**
3	Loops through all necessary substrings of `secret` (*no bounds errors*)	Responses **will not** earn the point if they skip overlapping occurrences	**1 point**
4	Counts number of identified occurrences of `guess` within `secret` (*in the context of a condition involving both* `secret` *and* `guess`)	Responses **can** still earn the point even if they • initialize count incorrectly or not at all • identify occurrences incorrectly	**1 point**
5	Calculates and returns correct final score (*algorithm*)	Responses **will not** earn the point if they • initialize count incorrectly or not at all • fail to use a loop • fail to compare `guess` to multiple substrings of `secret` • count the same matching substring more than once • use a changed or incorrect `guess` length when computing the score	**1 point**
		Total for part (a)	**5 points**

(b) `findBetterGuess`

	Scoring Criteria	Decision Rules	
6	Calls `scoreGuess` to get scores for `guess1` and `guess2`	Responses **will not** earn the point if they • fail to include parameters in the method calls • call the method on an object or class other than `this`	**1 point**
7	Compares the scores	Responses **will not** earn the point if they • only compare using `==` or `!=` • fail to use the result of the comparison in a conditional statement	**1 point**
8	Determines which of `guess1` and `guess2` is alphabetically greater	Responses **can** still earn the point even if they reverse the comparison Responses **will not** earn the point if they • reimplement `compareTo` incorrectly • use result of `compareTo` as if `boolean`	**1 point**
9	Returns the identified `guess1` or `guess2` (*algorithm*)	Responses **can** still earn the point even if they • call `scoreGuess` incorrectly • compare strings incorrectly Responses **will not** earn the point if they • reverse a comparison • omit either comparison • fail to return a guess in some case	**1 point**
		Total for part (b)	**4 points**

Question-specific penalties

None

Total for question 1 **9 points**

2011.4

Question 4: Cipher

Part (a):

```
private void fillBlock(String str) {
  int pos = 0;
  for (int r = 0; r < this.numRows; r++ ) {
    for (int c = 0; c < this.numCols; c++ ) {
      if (pos < str.length()) {
        this.letterBlock[r][c] = str.substring(pos, pos+1);
        pos++;
      } else {
        this.letterBlock[r][c] = "A";
      }
    }
  }
}
```

// Alternative solution

```
private void fillBlock(String str) {
  for (int r = 0; r < this.numRows; r++ ) {
    for (int c = 0; c < this.numCols; c++ ){
      if (str.length() > (c + (r * this.numCols))) {
        this.letterBlock[r][c] = str.substring(c + r * this.numCols,
                                               1 + c + r * this.numCols);
      } else {
        this.letterBlock[r][c] = "A";
      }
    }
  }
}
```

Question 4: Cipher (continued)

Part (b):

```
public String encryptMessage(String message) {
  String encryptedMessage = "";
  int chunkSize = this.numRows * this.numCols;
  while (message.length() > 0) {
    if (chunkSize > message.length()) {
      chunkSize = message.length();
    }
    fillBlock(message);
    encryptedMessage += encryptBlock();
    message = message.substring(chunkSize);
  }
  return encryptedMessage;
}
```

// Alternative solution

```
public String encryptMessage(String message) {
  if (message.length() == 0) return "";
  fillBlock(message);
  if (message.length() <= this.numRows * this.numCols) {
    return encryptBlock();
  }
  return (encryptBlock() +
          encryptMessage(message.substring(this.numRows * this.numCols)));
}
```

Question 3: PhraseEditor

Part (a) `replaceNthOccurrence` **5 points**

Intent: *Replace the nth occurrence of a given string with a given replacement*

+1 Calls `findNthOccurrence` to find the index of the `nth` occurrence

+1 Preserves `currentPhrase` only if `nth` occurrence does not exist

+1 Identifies components of `currentPhrase` to retain (uses `substring` to extract before/after)

+1 Creates replacement string using identified components and `repl`

+1 Assigns replacement string to instance variable (`currentPhrase`)

Part (b) `findLastOccurrence` **4 points**

Intent: *Return the index of the last occurrence of a given string*

+1 Calls `findNthOccurrence` to find the index of the *n*th occurrence

+1 Increments (or decrements) the value used as *n* when finding *n*th occurrence

+1 Returns the index of the last occurrence, if it exists

+1 Returns -1 only when no occurrences exist

Question-Specific Penalties

-1 (q) Uses `currentPhrase.findNthOccurrence`

-2 (r) Confused identifier instead of `currentPhrase`

Question 3: Scoring Notes

Part (a) `replaceNthOccurrence`			**5 points**
Points	Rubric Criteria	Responses earn the point if they ...	Responses will not earn the point if they ...
+1	Calls `findNthOccurrence` to find the index of the `nth` occurrence	● do not use the result of calling `findNthOccurrence`	
+1	Preserves `currentPhrase` only if `nth` occurrence does not exist		● fail to use a conditional
+1	Identifies components of `currentPhrase` to retain (uses `substring` to extract before/after)	● identify start and end of substring to be replaced	
+1	Creates replacement string using identified components and `repl`		● create a replacement string that is out of order
+1	Assigns replacement string to instance variable (`currentPhrase`)		
Part (b) `findLastOccurrence`			**4 points**
Points	Rubric Criteria	Responses earn the point if they ...	Responses will not earn the point if they ...
+1	Calls `findNthOccurrence` to find the index of the *n*th occurrence	● do not use the result of calling `findNthOccurrence`	● `return currentPhrase.lastIndexOf(str);` ● call `findNthOccurrence` with an integer parameter of 0
+1	Increments (or decrements) the value used as *n* when finding *n*th occurrence	● `return currentPhrase.lastIndexOf(str);` ● advance through `currentPhrase` searching for *n*th occurrence of `str`	
+1	Returns the index of the last occurrence, if it exists	● `return currentPhrase.lastIndexOf(str);` ● compute the correct value to be returned in all cases, but no return statement exists for any case	● shorten string being searched ● always return in first iteration of the loop
+1	Returns -1 only when no occurrences exist	● `return currentPhrase.lastIndexOf(str);`	● compute the correct value to be returned in all cases, but no return statement exists for any case ● always return in first iteration of the loop

2017.3

Question 3: PhraseEditor

Part (a)

```
public void replaceNthOccurrence(String str, int n, String repl)
{
      int loc = findNthOccurrence(str, n);

      if (loc != -1)
      {
            currentPhrase = currentPhrase.substring(0, loc) + repl +
                            currentPhrase.substring(loc + str.length());
      }
}
```

Part (b)

```
public int findLastOccurrence(String str)
{
      int n = 1;
      while (findNthOccurrence(str, n+1) != -1)
      {
            n++;
      }
      return findNthOccurrence(str, n);
}
```

Class FRQ Answer

Question 2: Step Tracker

Class:	`StepTracker`	**9 points**

Intent: *Define implementation of a class to record fitness data*

+1 Declares all appropriate `private` instance variables

+2 Constructor

+1 Declares header: `public StepTracker(int ___)`

+1 Uses parameter and appropriate values to initialize instance variables

+3 `addDailySteps` method

+1 Declares header: `public void addDailySteps(int ___)`

+1 Identifies active days and increments count

+1 Updates other instance variables appropriately

+1 `activeDays` method

+1 Declares and implements `public int activeDays()`

+2 `averageSteps` method

+1 Declares header: `public double averageSteps()`

+1 Returns calculated `double` average number of steps

Question 2: Scoring Notes

Class `StepTracker`			**9 points**
Points	Rubric Criteria	Responses earn the point even if they…	Responses will not earn the point if they…
+1	Declares all appropriate `private` instance variables		• omit keyword `private` • declare variables outside the class
+2	Constructor		
+1	Declares header: `public StepTracker(int ____)`	• omit keyword `public`	• declare method `private`
+1	Uses parameter and appropriate values to initialize instance variables	• initialize primitive instance variables to default values when declared	• fail to use the parameter to initialize some instance variable • fail to declare instance variables • initialize local variables instead of instance variables • assign variables to parameters
+3	`addDailySteps` method		
+1	Declares header: `public void addDailySteps(int ___)`	• omit keyword `public`	• declare method `private`
+1	Identifies active days and increments count	• put valid comparison erroneously in some other method	• fail to use the parameter as part of the comparison • fail to increment a count of active days • fail to increment an instance variable • compare parameter to some numeric constant
+1	Updates other instance variables appropriately		• update another instance variable only on active days • update another instance variable inappropriately • fail to update appropriate instance variable • update a local variable
+1	`activeDays` method		
+1	Declares and implements `public int activeDays()`	• return appropriate count of active days where the instance variables were updated improperly in `addDailySteps` or `activeDays`	• declare method `private` • return value that is not the number of active days • fail to return a value

Question 2: Scoring Notes (continued)

Points	Rubric Criteria	Responses earn the point even if they…	Responses will not earn the point if they…
+2	`averageSteps` method		
+1	Declares header: `public double averageSteps()`	- omit keyword `public`	- declare method `private`
+1	Returns calculated `double` average number of steps	- maintain instance variables improperly but calculate appropriate average - fail to handle the special case where no days are tracked	- use integer division - calculate something other than steps divided by days - fail to return

2019(2).2

Step Tracker

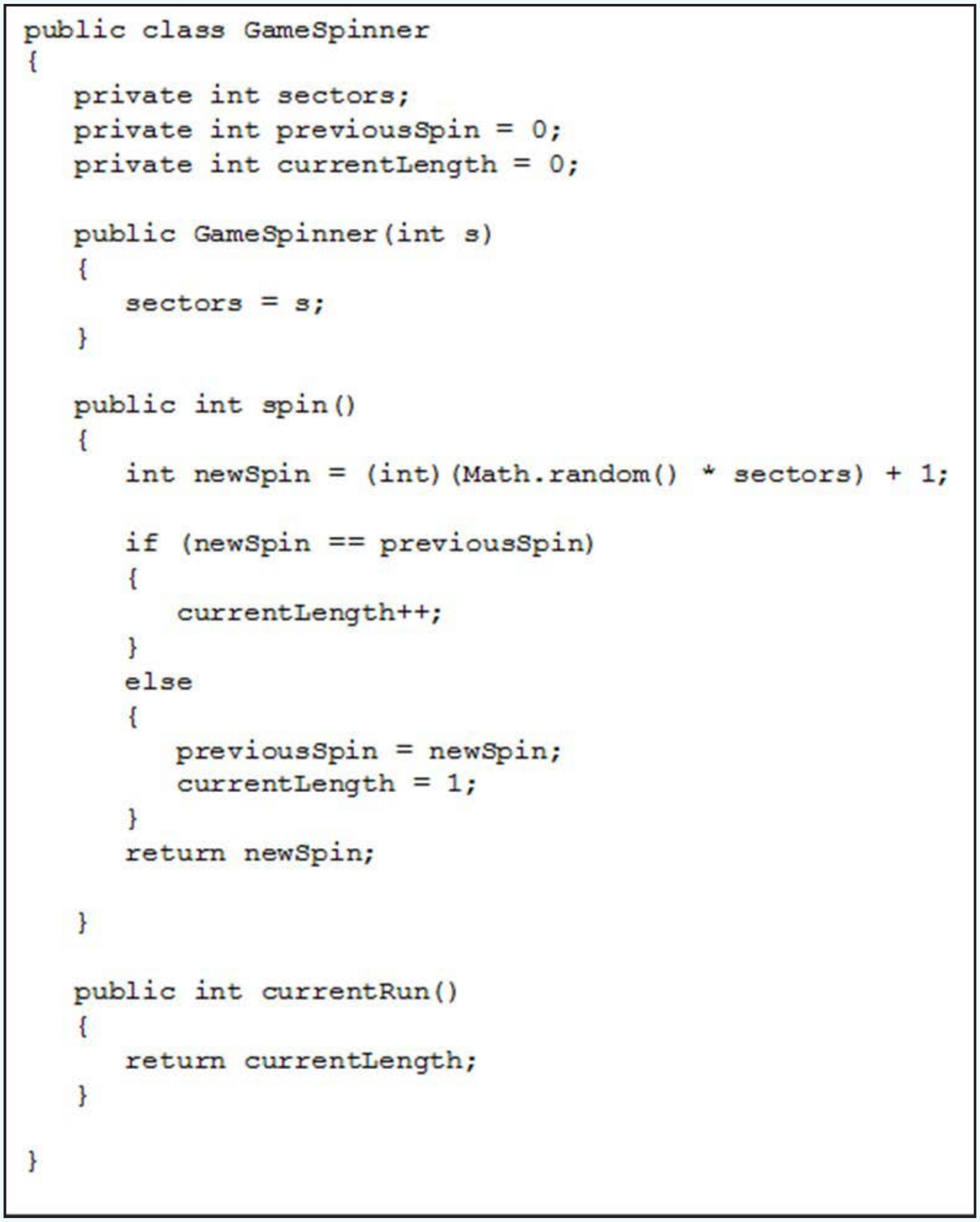
```
public class StepTracker
{
   private int minSteps;
   private int totalSteps;
   private int numDays;
   private int numActiveDays;

   public StepTracker(int threshold)
   {
      minSteps = threshold;
      totalSteps = 0;
      numDays = 0;
      numActiveDays = 0;
   }

   public void addDailySteps(int steps)
   {
      totalSteps += steps;
      numDays++;
      if (steps >= minSteps)
      {
         numActiveDays++;
      }
   }

   public int activeDays()
   {
      return numActiveDays;
   }

   public double averageSteps()
   {
      if (numDays == 0)
      {
         return 0.0;
      }
      else
      {
         return (double) totalSteps / numDays;
      }
   }
}
```

2019(1). 2

Question 2: Class Design 9 points

Learning Objectives: MOD-2.B MOD-2.D CON-1.B CON-1.D CON-2.A

Canonical solution



```
public class GameSpinner
{
   private int sectors;
   private int previousSpin = 0;
   private int currentLength = 0;

   public GameSpinner(int s)
   {
      sectors = s;
   }

   public int spin()
   {
      int newSpin = (int)(Math.random() * sectors) + 1;

      if (newSpin == previousSpin)
      {
         currentLength++;
      }
      else
      {
         previousSpin = newSpin;
         currentLength = 1;
      }
      return newSpin;

   }

   public int currentRun()
   {
      return currentLength;
   }

}
```

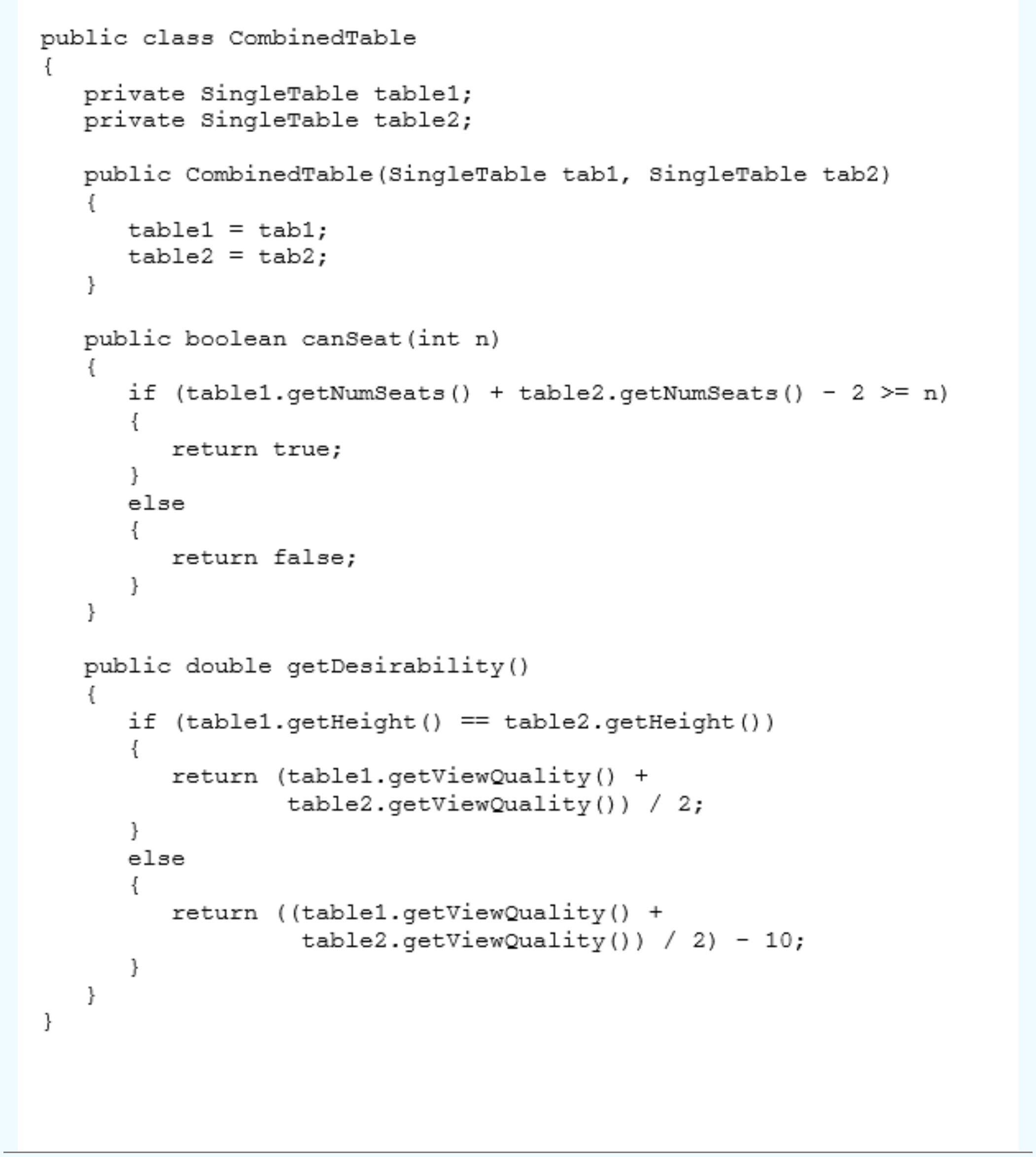
9 points

`GameSpinner`

	Scoring Criteria	Decision Rules	
1	Declares all appropriate `private` instance variables		1 point 3.B MOD-2.B
2	Declares method headers: `public int spin()` and `public int currentRun()`		1 point 3.B MOD-2.D
3	Declares header: `GameSpinner(int __)` (*must not be* `private`)		1 point 3.B MOD-2.B
4	Constructor initializes instance variable for number of sectors using parameter. Instance variables for previous spin and length of current run initialized correctly when declared or in constructor with default values.	**Responses still earn the point even if they...** • declare instance variables incorrectly.	1 point 3.B MOD-2.B
5	Computes random integer [1, number of sectors]		1 point 3.A CON-1.D
6	Compares new spin and last spin to determine required updates to state	**Responses still earn the point even if they...** • use an incorrectly computed random integer for new spin; or • incorrectly declare the instance variable intended to store last spin.	1 point 3.C CON-2.A
7	Updates instance variable that represents length of current run appropriately if new spin and previous spin are the same	**Responses still earn the point even if they...** • incorrectly compare new spin and last spin.	1 point 3.B MOD-2.D
8	Updates previous spin and length of current run appropriately when new spin differs from the previous spin	**Responses still earn the point even if they...** • incorrectly compare new spin and last spin.	1 point 3.C CON-1.B
9	`currentRun` returns updated instance variable value	**Responses still earn the point even if they...** • incorrectly update instance variables in the `spin` method.	1 point 3.B MOD-2.D
	Question-specific penalties		
	None		
		Total for question 2	**9 points**

Question 2: Class Design 9 points

Canonical solution



```
public class CombinedTable
{
   private SingleTable table1;
   private SingleTable table2;

   public CombinedTable(SingleTable tab1, SingleTable tab2)
   {
      table1 = tab1;
      table2 = tab2;
   }

   public boolean canSeat(int n)
   {
      if (table1.getNumSeats() + table2.getNumSeats() - 2 >= n)
      {
         return true;
      }
      else
      {
         return false;
      }
   }

   public double getDesirability()
   {
      if (table1.getHeight() == table2.getHeight())
      {
         return (table1.getViewQuality() +
                 table2.getViewQuality()) / 2;
      }
      else
      {
         return ((table1.getViewQuality() +
                  table2.getViewQuality()) / 2) - 10;
      }
   }
}
```

9 points

`CombinedTable`

	Scoring Criteria	Decision Rules	
1	Declares class header: `class CombinedTable` and constructor header: `CombinedTable(SingleTable ___, SingleTable ___)` (*must not be* `private`)	Responses **can** still earn the point even if they declare the class header as `class CombinedTable extends SingleTable`	**1 point**
2	Declares appropriate `private` instance variables including at least two `SingleTable` references	Responses **can** still earn the point even if they declare an additional instance variable to cache the number of seats at the combined table Responses **will not** earn the point if they • declare and initialize local variables in the constructor instead of instance variables • declare additional instance variable(s) that cache the desirability rating • omit keyword `private` • declare variables outside the class	**1 point**
3	Constructor initializes instance variables using parameters	Responses **can** still earn the point even if they declare and initialize local variables in the constructor instead of instance variables	**1 point**
4	Declares header: `public boolean canSeat(int ___)`		**1 point**
5	Calls `getNumSeats` on a `SingleTable` object	Responses **can** still earn the point even if they call `getNumSeats` on constructor parameters or local variables of type `SingleTable` in the constructor Responses **will not** earn the point if they call the `SingleTable` accessor method on something other than a `SingleTable` object	**1 point**
6	`canSeat(n)` returns `true` if and only if sum of seats of two tables `- 2 >= n`	Responses **can** still earn the point even if they call `getNumSeats` incorrectly	**1 point**
7	Declares header: `public double getDesirability()`		**1 point**
8	Calls `getHeight` and `getViewQuality` on `SingleTable` objects	Responses **can** still earn the point even if they call `getHeight` or `getViewQuality` on constructor parameters or local variables of type `SingleTable` in the constructor	**1 point**

		Responses **will not** earn the point if they call the `SingleTable` accessor methods on something other than a `SingleTable` object	
9	`getDesirability` computes average of constituent tables' view desirabilities	Responses **can** still earn the point even if they • call `getHeight` or `getViewQuality` on constructor parameters or local variables of type `SingleTable` in the constructor • fail to return the computed average (*return is not assessed*) Responses **will not** earn the point if they • fail to have an `if` statement and a correct calculation • choose the incorrect value (average vs. average – 10) based on evaluation of the `if` statement condition	**1 point**

Question-specific penalties

None

Total for question 2 9 points

Question 2: Guessing Game

Class:	`HiddenWord`	**9 points**

Intent: *Define implementation of class to represent hidden word in guessing game*

+1 Uses correct class, constructor, and method headers

+1 Declares appropriate `private` instance variable

+1 Initializes instance variable within constructor using parameter

+6 Implement `getHint`

- **+1** Accesses all letters in both guess and hidden word in loop (*no bounds errors in either*)

- **+4** Process letters within loop

  - **+1** Extracts and compares corresponding single letters from guess and hidden word

  - **+1** Tests whether guess letter occurs in same position in both guess and hidden word

  - **+1** Tests whether guess letter occurs in hidden word but not in same position as in guess

  - **+1** Adds correct character exactly once to the hint string based on the test result

- **+1** Declares, initializes, and returns constructed hint string

Question-Specific Penalties

-1 (t) Uses `get` to access letters from strings

-2 (u) Consistently uses incorrect name instead of instance variable name for hidden word

2015.2

Guessing Game

```
public class HiddenWord
{
    private String word;

    public HiddenWord(String hWord)
    {
        word = hWord;
    }

    public String getHint(String guess){
        String hint = "";
        for (int i = 0; i < guess.length(); i++){
          if (guess.substring(i,i+1).equals(word.substring(i,i+1))){
                hint += guess.substring(i,i+1);
          } else if (word.indexOf(guess.substring(i,i+1))!= -1){
                hint += "+";
          } else {
                hint += "*";
          }
        }
        return hint;
    }

}
```

Question 1: Random String Chooser

Part (a)	`RandomStringChooser`	**7 points**

Intent: *Define implementation of class to choose a random string*

- **+1** Uses correct class, constructor, and method headers
- **+1** Declares appropriate `private` instance variable(s)
- **+1** Initializes all instance variable(s) (*point lost if parameter not used in any initialization*)
- **+4** Implements `getNext`
  - **+1** Generates a random number in the proper range (*point lost for improper or missing cast*)
  - **+1** Chooses a string from instance variable using generated random number
  - **+1** Updates state appropriately (*point lost if constructor parameter is altered*)
  - **+1** Returns chosen string or `"NONE"` as appropriate

Part (b)	`RandomLetterChooser`	**2 points**

Intent: *Define implementation of a constructor of a class that extends* `RandomStringChooser`

- **+1** `getSingleLetters(str)`
- **+1** `super(getSingleLetters(str));` (*point lost if not first statement in constructor*)

2016.1

Random String Chooser

Part (a):

```
public class RandomStringChooser
{
    private List<String> words;

    public RandomStringChooser(String[] wordArray)
    {
      words = new ArrayList<String>();

      for (String singleWord : wordArray)
      {
           words.add(singleWord);
      }
    }

    public String getNext()
    {
      if (words.size() > 0)
      {
           return words.remove((int)(Math.random() * words.size()));
      }
      return "NONE";
    }
}
```

Part (b):

```
      public RandomLetterChooser(String str)
      {
           super(getSingleLetters(str));
      }
```

2010.2

```
public class APLine {
  /** State variables. Any numeric type; object or primitive. */
  private int a, b, c;

  /** Constructor with 3 int parameters. */
  public APLine(int a, int b, int c) {
    this.a = a;
    this.b = b;
    this.c = c;
  }

  /** Determine the slope of this APLine. */
  public double getSlope() {
    return ( - (this.a / (double) this.b));
  }

  /** Determine if coordinates represent a point on this APLine. */
  public boolean isOnLine(int x, int y) {
    return (0 == (this.a * x) + (this.b * y) + this.c);
  }
}


// Alternative solution (state variables of type double):

public class APLine {
  private double a1, b1, c1;

  public APLine(int a, int b, int c) {
    this.a1 = a;
    this.b1 = b;
    this.c1 = c;
  }

  public double getSlope() {
    return -(this.a1 / this.b1);
  }

  public boolean isOnLine(int x, int y) {
    return (0 == (this.a1 * x) + (this.b1 * y) + this.c1);
  }
}
```

ArrayList FRQ Answer

Question 3: Delimiters

Part (a)	`getDelimitersList`	**4 points**

Intent: *Store delimiters from an array in an* `ArrayList`

+1 Creates `ArrayList<String>`

+1 Accesses all elements in array `tokens` (*no bounds errors*)

+1 Compares strings in `tokens` with both instance variables (*must be in the context of a loop*)

+1 Adds delimiters into `ArrayList` in original order

Part (b)	`isBalanced`	**5 points**

Intent: *Determine whether open and close delimiters in an* `ArrayList` *are balanced*

+1 Initializes accumulator(s)

+1 Accesses all elements in `ArrayList delimiters` (*no bounds errors*)

+1 Compares strings in `delimiters` with instance variables and updates accumulator(s) accordingly

+1 Identifies and returns appropriate `boolean` value to implement one rule

+1 Identifies and returns appropriate `boolean` values for all cases

Question 3: Scoring Notes

Part (a) `getDelimitersList`			**4 points**
Points	Rubric Criteria	Responses earn the point even if they…	Responses will not earn the point if they…
+1	Creates `ArrayList<String>`	• omit `<String>`	• omit the keyword `new`
+1	Accesses all elements in array `tokens` (*no bounds errors*)	• return incorrectly inside the loop	• treat `tokens` as a single string • access elements of `tokens` as if from an `ArrayList` (e.g., `tokens.get(i)`)
+1	Compares strings in `tokens` with both instance variables (*must be in the context of a loop*)	• access elements of `tokens` as if from an `ArrayList` (e.g., `tokens.get(i)`)	• use `==` for string comparison • treat `tokens` as a single string
+1	Adds delimiters into `ArrayList` in original order	• add a delimiter by accessing `tokens` incorrectly (e.g., `tokens.get(i)`)	• add a token that is not a delimiter • do not maintain the original delimiter order

Part (b) `isBalanced`			**5 points**
Points	Rubric Criteria	Responses earn the point even if they…	Responses will not earn the point if they…
+1	Initializes accumulator(s)	• initialize inside the loop	• initialize an accumulator variable but don't update it
+1	Accesses all elements in `ArrayList` `delimiters` (*no bounds errors*)	• return incorrectly inside the loop	• access elements of `delimiters` as if from an array (e.g., `delimiters[i]`)
+1	Compares strings in `delimiters` with instance variables and updates accumulator(s) accordingly	• access elements of `delimiters` as if from an array (e.g., `delimiters[i]`)	• use `==` for string comparison • adjust an accumulator without a guarding condition
+1	Identifies and returns appropriate `boolean` value to implement one rule	• check for more closing delimiters (inside a loop) and return `false` • return `true` if the number of open and close delimiters is the same, and `false` otherwise (after a loop)	
+1	Identifies and returns appropriate `boolean` values for all cases	• have correct logic with the exception of a loop bounds error, accessing elements as if from an array, or using `==` for string comparison	• initialize accumulator inside a loop • fail to check for more closing delimiters inside a loop

2019(2).3

Question 3: Delimiters

Part (a)

```
public ArrayList<String> getDelimitersList(String[] tokens)
{
   ArrayList<String> d = new ArrayList<String>();
   for (String str : tokens)
   {
      if (str.equals(openDel) || str.equals(closeDel))
      {
         d.add(str);
      }
   }
   return d;
}
```

Part (b)

```
public boolean isBalanced(ArrayList<String> delimiters)
{
   int openCount = 0;
   int closeCount = 0;

   for (String str : delimiters)
   {
      if (str.equals(openDel))
      {
         openCount++;
      }
      else
      {
         closeCount++;
      }

      if (closeCount > openCount)
      {
         return false;
      }
   }

   if (openCount == closeCount)
   {
      return true;
   }
   else
   {
      return false;
   }
}
```

2010.1

Question 1: Master Order

Part (a):

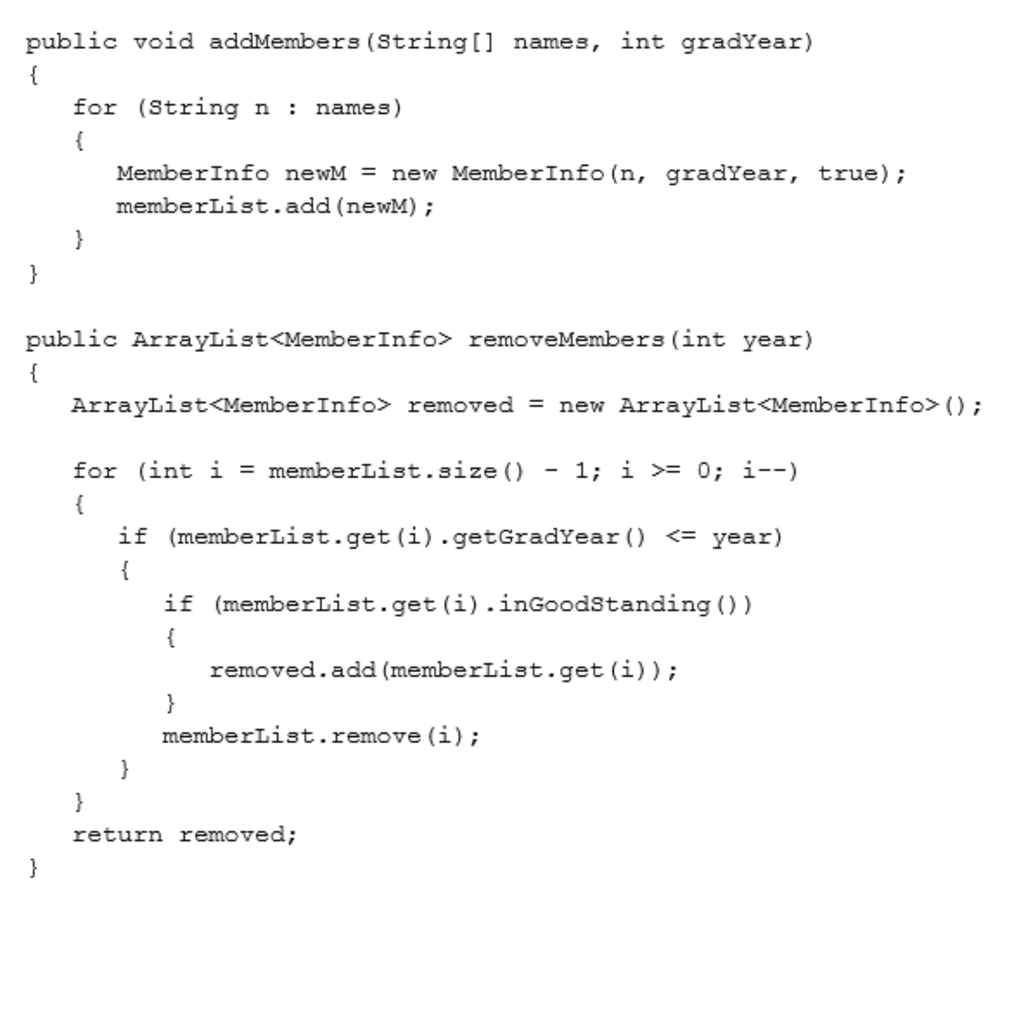
```
public int getTotalBoxes() {
  int sum = 0;
  for (CookieOrder co : this.orders) {
    sum += co.getNumBoxes();
  }
  return sum;
}
```

Part (b):

```
public int removeVariety(String cookieVar) {
  int numBoxesRemoved = 0;
  for (int i = this.orders.size() - 1; i >= 0; i--) {
    if (cookieVar.equals(this.orders.get(i).getVariety())) {
      numBoxesRemoved += this.orders.get(i).getNumBoxes();
      this.orders.remove(i);
    }
  }
  return numBoxesRemoved;
}

// Alternative solution (forward traversal direction):

public int removeVariety(String cookieVar) {
  int numBoxesRemoved = 0;
  int i = 0;
  while (i < this.orders.size()) {
    if (cookieVar.equals(this.orders.get(i).getVariety())) {
      numBoxesRemoved += this.orders.get(i).getNumBoxes();
      this.orders.remove(i);
    } else {
      i++;
    }
  }
  return numBoxesRemoved;
}
```

Question 3: Array / ArrayList 9 points

Canonical solution



(a) **3 points**

```
public void addMembers(String[] names, int gradYear)
{
   for (String n : names)
   {
      MemberInfo newM = new MemberInfo(n, gradYear, true);
      memberList.add(newM);
   }
}
```

(b) **6 points**

```
public ArrayList<MemberInfo> removeMembers(int year)
{
   ArrayList<MemberInfo> removed = new ArrayList<MemberInfo>();

   for (int i = memberList.size() - 1; i >= 0; i--)
   {
      if (memberList.get(i).getGradYear() <= year)
      {
         if (memberList.get(i).inGoodStanding())
         {
            removed.add(memberList.get(i));
         }
         memberList.remove(i);
      }
   }
   return removed;
}
```

(a) `addMembers`

	Scoring Criteria	Decision Rules	
1	Accesses all elements of `names` (*no bounds errors*)	Responses **will not** earn the point if they fail to access elements of the array, even if loop bounds are correct	**1 point**
2	Instantiates a `MemberInfo` object with name from array, provided year, and good standing		**1 point**
3	Adds `MemberInfo` objects to `memberList` (*in the context of a loop*)	Responses **can** earn the point even if they instantiate `MemberInfo` objects incorrectly	**1 point**
		Total for part (a)	**3 points**

(b) `removeMembers`

	Scoring Criteria	Decision Rules	
4	Declares and initializes an `ArrayList` of `MemberInfo` objects	Responses **will not** earn the point if they initialize the variable with a reference to the instance variable	**1 point**
5	Accesses all elements of `memberList` for potential removal (*no bounds errors*)	Responses **will not** earn the point if they • fail to use `get(i)` • fail to attempt to remove an element • skip an element • throw an exception due to removing	**1 point**
6	Calls `getGradYear` or `inGoodStanding`	Responses **can** still earn the point even if they call only one of the methods Responses **will not** earn the point if they • ever include parameters in either method call • ever call either method on an object other than `MemberInfo`	**1 point**
7	Distinguishes any three cases, based on graduation status and standing	Responses **will not** earn the point if they fail to behave differently in all three cases	**1 point**
8	Identifies graduating members	Responses **can** still earn the point even if they • fail to distinguish three cases • fail to access standing at all • access the graduating year incorrectly Responses **will not** earn the point if they confuse `<` and `<=` in the comparison	**1 point**
9	Removes appropriate members from `memberList` and adds appropriate members to the `ArrayList` to be returned	Responses **can** still earn the point even if they • call `getGradYear` or `inGoodStanding` incorrectly • access elements of `memberList` incorrectly • initialize the `ArrayList` incorrectly • fail to return the list that was built (*return is not assessed*) Responses **will not** earn the point if they • fail to declare an `ArrayList` to return • fail to distinguish the correct three cases, with the exception of confusing the `<` and `<=` in the comparison	**1 point**
		Total for part (b)	**6 points**

Question-specific penalties
None

Total for question 3 9 points

2018.2

Question 2: Word Pair

Part (a)	`WordPairList`	**5 points**

Intent: *Form pairs of strings from an array and add to an* `ArrayList`

- **+1** Creates new `ArrayList` and assigns to `allPairs`
- **+1** Accesses all elements of `words` (*no bounds errors*)
- **+1** Constructs new `WordPair` using distinct elements of `words`
- **+1** Adds all necessary pairs of elements from word array to `allPairs`
- **+1** **On exit:** `allPairs` contains all necessary pairs and no unnecessary pairs

Part (b)	`numMatches`	**4 points**

Intent: *Count the number of pairs in an* `ArrayList` *that have the same value*

- **+1** Accesses all elements in `allPairs` (*no bounds errors*)
- **+1** Calls `getFirst` or `getSecond` on an element from list of pairs
- **+1** Compares first and second components of a pair in the list
- **+1** Counts number of matches of pair-like values

Question-Specific Penalties

- **-1** (z) Constructor returns a value

Question 2: Scoring Notes

Part (a) `WordPairList`			**5 points**
Points	Rubric Criteria	Responses earn the point if they...	Responses will not earn the point if they...
+1	Creates new `ArrayList` and assigns to `allPairs`	• `allPairs = new ArrayList();` • `allPairs = new ArrayList<>();` • `this.allPairs = ...`	• initialize a local variable that is never assigned to `allPairs`
+1	Accesses all elements of `words` (*no bounds errors*)		
+1	Constructs new `WordPair` using distinct elements of `words`		
+1	Adds all necessary pairs of elements from word array to `allPairs`	• have a loop bounds error • add unnecessary pairs	• improperly add to an `ArrayList`, e.g., `allPairs.get(i) = x;` • only add consecutive pairs `(words[i], words[i+1])`
+1	**On exit:** `allPairs` contains all necessary pairs and no unnecessary pairs	• improperly add to an `ArrayList`, e.g., `allPairs.get(i) = x;` • have a loop bounds error	• add pairs (i, i) or (i, j) where i > j
Part (b) `numMatches`			**4 points**
Points	Rubric Criteria	Responses earn the point if they...	Responses will not earn the point if they...
+1	Accesses all elements in `allPairs` (*no bounds errors*)		• access elements of `allPairs` as array elements (e.g., `allPairs[i]`)
+1	Calls `getFirst` or `getSecond` on an element from list of pairs		
+1	Compares first and second components of a pair in the list		• compare using ==
+1	Counts number of matches of pair-like values		• fail to initialize the counter

Return is not assessed in part (b).

2018.2

Word Pair

Part (a)

```
public WordPairList(String[] words)
{
   allPairs = new ArrayList<WordPair>();

   for (int i = 0; i < words.length-1; i++)
   {
      for (int j = i+1; j < words.length; j++)
      {
         allPairs.add(new WordPair(words[i], words[j]));
      }
   }
}
```

Part (b)

```
public int numMatches()
{
   int count = 0;

   for (WordPair pair: allPairs)
   {
      if (pair.getFirst().equals(pair.getSecond()))
      {
         count++;
      }
   }
   return count;
}
```

2012.1

Climbing Club

Part (a):

```
public void addClimb(String peakName, int climbTime) {
   this.climbList.add(new ClimbInfo(peakName, climbTime));
}
```

Part (b):

```
public void addClimb(String peakName, int climbTime) {
   for (int i = 0; i < this.climbList.size(); i++) {
      if (peakName.compareTo(this.climbList.get(i).getName()) <= 0) {
         this.climbList.add(i, new ClimbInfo(peakName, climbTime));
         return;
      }
   }
   this.climbList.add(new ClimbInfo(peakName, climbTime));
}
```

Part (c):

NO

YES

2013.1

SongList

Part (a):

```
public DownloadInfo getDownloadInfo(String title) {
      for (DownloadInfo info : downloadList){
           if (info.getTitle().equals(title)){
              return info;
           }
      }
      return null;
}
```

Part (b):

```
public void updateDownloads(List<String> titles) {
    for (String title : titles) {
        DownloadInfo foundInfo = getDownloadInfo(title);
         if (foundInfo == null){
           downloadList.add(new DownloadInfo(title));
        }
        else {
            foundInfo.incrementTimesDownloaded();
        }
    }
}
```

Question 1: Digits

Part (a) `Digits` constructor **5 points**

Intent: *Initialize instance variable using passed parameter*

+1 Constructs `digitList`

+1 Identifies a digit in `num`

+1 Adds at least one identified digit to a list

+1 Adds all identified digits to a list (*must be in context of a loop)*

+1 **On exit:** `digitList` contains all and only digits of `num` in the correct order

Part (b) `isStrictlyIncreasing` **4 points**

Intent: *Determine whether or not elements in* `digitList` *are in increasing order*

+1 Compares at least one identified consecutive pair of `digitList` elements

+1 Determines if a consecutive pair of `digitList` is out of order (*must be in context of a* `digitList` *traversal*)

+1 Compares all necessary consecutive pairs of elements (*no bounds errors*)

+1 Returns `true` iff all consecutive pairs of elements are in order; returns `false` otherwise

Question-Specific Penalties

-2 (q) Uses confused identifier instead of `digitList`

Question 1: Scoring Notes

Part (a) `Digits` constructor			**5 points**
Points	Rubric Criteria	Responses earn the point if they ...	Responses will not earn the point if they ...
+1	Constructs `digitList`		• initialize a local variable instead of `digitList` • create an `ArrayList<int>`
+1	Identifies a digit in `num`	• identify one digit of `num` or a length one substring/character of the `String` representation of `num`	• treat `num` itself as a `String` • convert `num` to a `String` incorrectly
+1	Adds at least one identified digit to a list	• call `add` for some `ArrayList` using the previously identified digit, even if that digit was identified incorrectly	• add `String` or `char` to `digitList` without proper conversion to the correct type
+1	Adds all identified digits to a list (*must be in the context of a loop*)	• call `add` for some `ArrayList` using previously identified digits, even if those digits were identified incorrectly	• identify only 1 digit
+1	**On exit:** `digitList` contains all and only digits of `num` in the correct order	• add to `digitList` even if it is not instantiated properly	• obtain a list with the digits in reverse order • omit one or more digits • add extra digits • mishandle edge case, e.g., 0 or 10 • make a bounds error processing the `String` representation of `num`
Part (b) `isStrictlyIncreasing`			**4 points**
Points	Rubric Criteria	Responses earn the point if they ...	Responses will not earn the point if they ...
+1	Compares at least one identified consecutive pair of `digitList` elements	• compare two consecutive `Integers` using `compareTo` • explicitly convert two consecutive `Integers` to `ints` and compare those with >=, <= etc. • use auto-unboxing to convert two consecutive `Integers` to `ints` and compare those with >=, <= etc.	• access `digitList` as an array or string • fail to call `.get()` • compare using `!>`
+1	Determines if a consecutive pair of `digitList` is out of order (*must be in context of a* `digitList` *traversal*)	• determine the correct relationship between the two compared consecutive elements, even if the syntax of the comparison is incorrect	• fail to consider the case where the two elements are equal for the false case
+1	Compares all necessary consecutive pairs of elements (*no bounds errors*)		• return early
+1	Returns `true` iff all consecutive pairs of elements are in order; returns `false` otherwise	• compare consecutive pairs for inequality, but fail to consider the case when two elements are equal	• return prematurely via `if (...) return false; else return true;`

Digits

Part (a)

```
public Digits(int num)
{
      digitList = new ArrayList<Integer>();

      if (num == 0)
      {
            digitList.add(new Integer(0));
      }

      while (num > 0)
      {
            digitList.add(0, new Integer(num % 10));
            num /= 10;
      }
}
```

Part (b)

```
public boolean isStrictlyIncreasing()
{
      for (int i = 0; i < digitList.size()-1; i++)
      {
            if (digitList.get(i).intValue() >= digitList.get(i+1).intValue())
            {
                  return false;
            }
      }
      return true;
}
```

Note: The solutions shown above were written in compliance with the AP Java subset methods listed for `Integer` objects. Students were allowed to use the automatic "boxing" and "unboxing" of `Integer` objects in their solutions, which eliminates the need to use "`new Integer(...)`" in part (a) and "`intValue()`" in part (b).

2009.4

Tile Game

PART A:

```
/** Determines where to insert tile,
 *  in its current orientation, into game board
 *  @param tile the tile to be placed on the game board
 *  @return the position of tile where tile is to be inserted:
 *        0 if the board is empty;
 *       -1 if tile does not fit in front, at end,
 *            or between any existing tiles;
 *        otherwise, 0 ≤ position returned ≤ board.size()
 */
private int getIndexForFit(NumberTile tile)
{
  if ((this.board.size() == 0) ||
      (tile.getRight() == this.board.get(0).getLeft()))
    return 0;
  for (int i = 1; i < this.board.size(); i++)
  {
    if (tile.getLeft() == this.board.get(i-1).getRight() &&
        tile.getRight() == this.board.get(i).getLeft())
      return i;
  }
  if (tile.getLeft() == this.board.get(this.board.size() - 1).getRight())
    return this.board.size();

  return -1;
}
```

Question 4: Tile Game (continued)

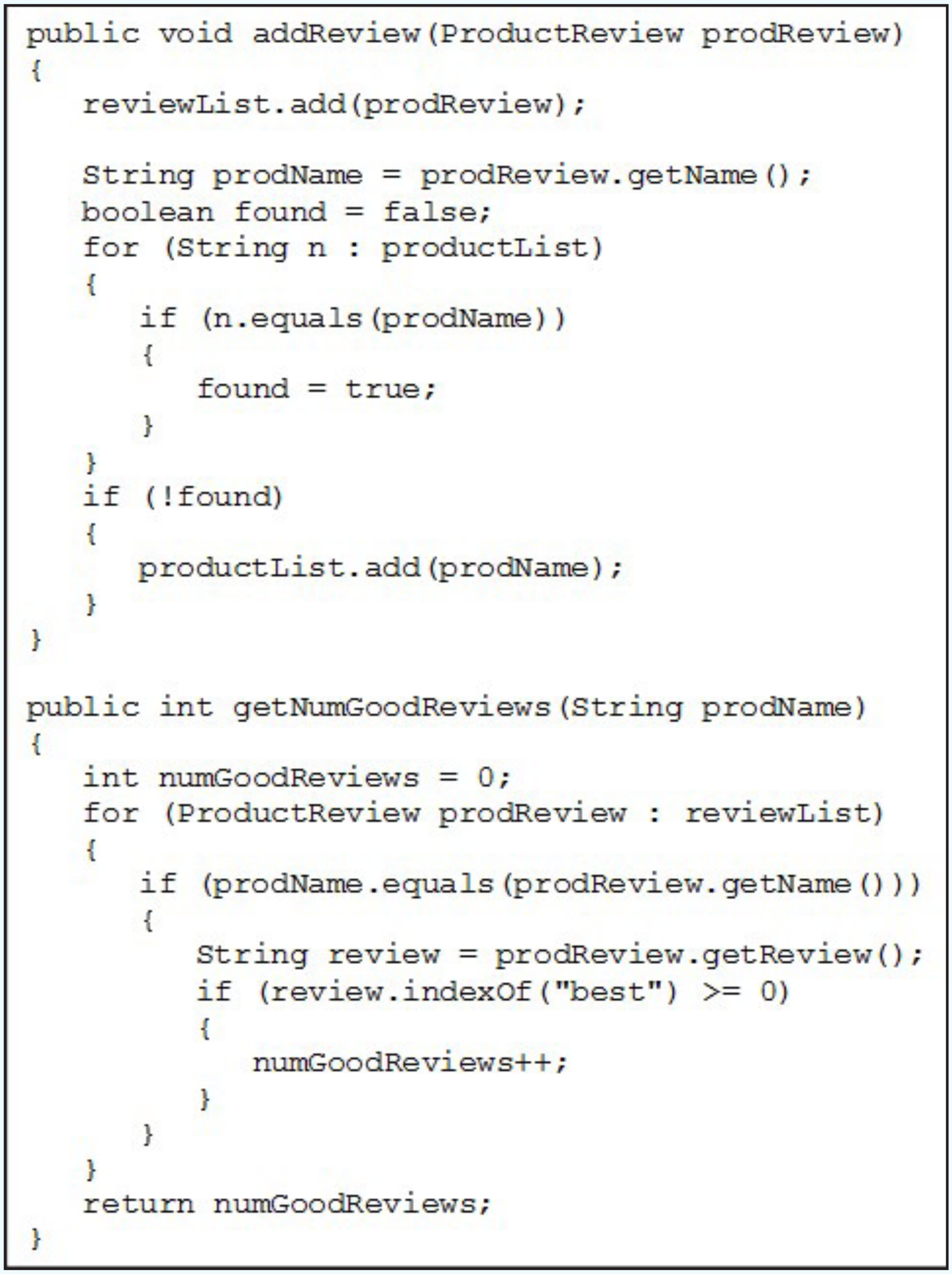
PART B:

```
/** Places tile on the game board if it fits (checking all possible
 *    tile orientations if necessary).
 *  If there are no tiles on the game board,
 *    the tile is placed at position 0.
 *  The tile should be placed at most 1 time.
 *  Precondition: board is not null
 *  @param tile the tile to be placed on the game board
 *  @return true if tile is placed successfully; false otherwise
 *  Postcondition: the orientations of the other tiles on the board
 *                 are not changed
 *  Postcondition: the order of the other tiles on the board
 *                 relative to each other is not changed
 */
public boolean insertTile(NumberTile tile)
{
  int index = getIndexForFit(tile);
  int test = 1;
  while (index == -1 && test < 4)
  {
    tile.rotate();
    index = getIndexForFit(tile);
    test++;
  }
  if (index != -1)
    this.board.add(index, tile);

  return (index != -1);
}
```

Question 3: Array/ArrayList 9 points

Learning Objectives: VAR-1.E.b VAR-2.D VAR-2.E.a MOD-1.G CON-2.F.a CON-2.J.a CON-2.K

Canonical solution



(a) **6 points**

```
public void addReview(ProductReview prodReview)
{
   reviewList.add(prodReview);

   String prodName = prodReview.getName();
   boolean found = false;
   for (String n : productList)
   {
      if (n.equals(prodName))
      {
         found = true;
      }
   }
   if (!found)
   {
      productList.add(prodName);
   }
}
```

(b) **3 points**

```
public int getNumGoodReviews(String prodName)
{
   int numGoodReviews = 0;
   for (ProductReview prodReview : reviewList)
   {
      if (prodName.equals(prodReview.getName()))
      {
         String review = prodReview.getReview();
         if (review.indexOf("best") >= 0)
         {
            numGoodReviews++;
         }
      }
   }
   return numGoodReviews;
}
```

(a) `addReview`

	Scoring Criteria	Decision Rules	
1	Adds a `ProductReview` object to `reviewList`	**Responses still earn the point even if they...** • add a `ProductReview` object other than the one referenced by the parameter `prodReview`.	**1 point** 3.D VAR-2.D
2	Gets product name of review to be added		**1 point** 3.A MOD-1.G
3	Traverses `productList` (*no bounds errors*)	**Responses still earn the point even if they...** • use a `for`, an enhanced `for`, or a `while` loop.	**1 point** 3.D VAR-2.E.a
4	Compares name in `productList` with name from review to be added	**Responses still earn the point even if they...** • use an incorrectly accessed value for either name.	**1 point** 3.C VAR-1.E.b
5	Adds new product name to `productList`	**Responses still earn the point even if they...** • add the new product name under the wrong conditions; or • add an incorrectly accessed value for the new product name	**1 point** 3.D VAR-2.D
6	Correctly adds product name to `productList` if and only if the product name is not already in `productList`		**1 point** 3.D CON-2.K
		Total for part (a)	**6 points**

(b) `getNumGoodReviews`

	Scoring Criteria	Decision Rules	
7	Traverses `reviewList` (*no bounds errors*)	**Responses still earn the point even if they...** • use a `for`, an enhanced `for`, or a `while` loop.	**1 point** 3.D VAR-2.E.a
8	Selects all and only reviews with matching product names that contain `"best"`		**1 point** 3.C CON-2.F.a
9	Returns correct count of good reviews		**1 point** 3.D CON-2.J.a
		Total for part (b)	**3 points**

Question-specific penalties

None

Total for question 3 **9 points**

2007.3

Answer Sheets

PART A:

```
public double getScore(ArrayList<String> key)
{
    double score = 0.0;
    for (int i = 0; i < answers.size(); i++) {
        if (answers.get(i).equals(key.get(i))) {
            score += 1.0;
        }
        else if (!answers.get(i).equals("?")) {
            score -= 0.25;
        }
    }
    return score;
}
```

PART B:

```
public String highestScoringStudent(ArrayList<String> key)
{
    StudentAnswerSheet highest = sheets.get(0);
    for (StudentAnswerSheet sheet : sheets) {
        if (sheet.getScore(key) > highest.getScore(key)) {
            highest = sheet;
        }
    }
    return highest.getName();
}
```

2008.2

String Coder

PART A:

```
public String decodeString(ArrayList<StringPart> parts)
{
  String expanded = "";
  for (StringPart nextPart : parts)
  {
    int ending = nextPart.getStart()+nextPart.getLength();
    expanded += masterString.substring(nextPart.getStart(), ending);
  }
  return expanded;
}
```

PART B:

```
public ArrayList<StringPart> encodeString(String word)
{
  ArrayList<StringPart> parts = new ArrayList<StringPart>();

  while (word.length() > 0)
  {
    StringPart nextPart = findPart(word);
    parts.add(nextPart);
    word = word.substring(nextPart.getLength());
  }
  return parts;
}
```

ALTERNATE SOLUTION:

```
public ArrayList<StringPart> encodeString(String word)
{
  ArrayList<StringPart> parts = new ArrayList<StringPart>();

  int index = 0;
  while (index < word.length())
  {
    StringPart nextPart = findPart(word.substring(index));
    parts.add(nextPart);
    index += nextPart.getLength();
  }
  return parts;
}
```

2016.4

String Formatter

Part (a) `totalLetters` **2 points**

Intent: *Calculate the total number of letters in a list of words*

+1 Accesses all strings in `wordList` and adds length of each to accumulated count (*no bounds errors*)

+1 Initializes and returns accumulated count

Part (b) `basicGapWidth` **2 points**

Intent: *Calculate the minimum number of spaces (basic gap width) to be placed between each word in the formatted string*

+1 Calls `totalLetters` correctly and uses result

+1 Returns correct calculated value

Part (c) `format` **5 points**

Intent: *Return a formatted string consisting of words from* `wordList` *separated by one or more spaces*

+1 Calls `basicGapWidth` and `leftoverSpaces` correctly and uses results

+1 Adds all strings in `wordList` to formatted string in original order (*no bounds errors*)

+1 Inserts *basicGapWidth* spaces between each pair of words in formatted string

+1 Inserts one space between first *leftoverSpaces* pairs of words in formatted string

+1 Initializes and returns formatted string (*no extra or deleted characters*)

Question 4: String Formatter

Part (a):

```
public static int totalLetters(List<String> wordList)
{
      int total = 0;

      for (String word : wordList)
      {
            total += word.length();
      }
      return total;
}
```

Part (b):

```
public static int basicGapWidth(List<String> wordList, int formattedLen)
{
      return (formattedLen – totalLetters(wordList)) / (wordList.size()-1);
}
```

Part (c):

```
public static String format(List<String> wordList, int formattedLen)
{
      String formatted = "";
      int gapWidth = basicGapWidth(wordList, formattedLen);
      int leftovers = leftoverSpaces(wordList, formattedLen);

      for (int w = 0; w < wordList.size() - 1; w++)
      {
            formatted = formatted + wordList.get(w);
            for (int i = 0; i < gapWidth; i++)
            {
                  formatted = formatted + " ";
            }
            if (leftovers > 0)
            {
                  formatted = formatted + " ";
                  leftovers--;
            }
      }
      formatted = formatted + wordList.get(wordList.size() - 1);

      return formatted;
}
```

Question 2: Log Messages

Part (a) `LogMessage` constructor **2 points**

Intent: *Initialize instance variables using passed parameter*

+1 Locates colon

+1 Initializes instance variables with correct parts of the parameter

Part (b) `containsWord` **2 points**

Intent: *Determine whether description properly contains a keyword*

+1 Identifies at least one properly-contained occurrence of `keyword in description`

+1 Returns `true` if and only if `description` properly contains `keyword`
Returns `false` otherwise (*no bounds errors*)

Part (c) `removeMessages` **5 points**

Intent: *Remove log messages containing keyword from system log list and return these messages in a new list*

+1 Accesses all items in `messageList` (*no bounds errors; point lost if no removal attempted*)

+1 Identifies keyword-containing entry using `containsWord`

+1 Adds all and only identified entries to new list (*point lost if original order not maintained*)

+1 Removes all identified entries from `messageList` (*point lost if* `messageList` *reordered)*

+1 Constructs and returns `new ArrayList<LogMessage>`

2016.2

2: Log Messages

Part (a):

```
public LogMessage(String message)
{
      int colon = message.indexOf(":");
      machineId = message.substring(0, colon);
      description = message.substring(colon + 1);
}
```

Part (b):

```
public boolean containsWord(String keyword)
{
   if (description.equals(keyword))
   {   return true;   }
   if (description.indexOf(keyword + " ") == 0)
   {   return true;   }
   if (description.indexOf(" " + keyword + " ") != -1)
   {   return true;   }
   if (description.length() > keyword.length())
   {
       if ((description.substring(description.length() -
                                  keyword.length() - 1).equals(
                                  " " + keyword)))
       {
          return true;
       }
   }
   return false;
}
```

Part (c):

```
public List<LogMessage> removeMessages(String keyword)
{
      List<LogMessage> removals = new ArrayList<LogMessage>();

      for (int i = 0; i < messageList.size(); i++)
      {
           if (messageList.get(i).containsWord(keyword))
           {
                removals.add(messageList.remove(i));
                i--;
           }
      }
      return removals;
}
```

2006.1

Daily Schedule

PART A:

```
public boolean conflictsWith(Appointment other)
{
  return getTime().overlapsWith(other.getTime());
}
```

PART B:

```
public void clearConflicts(Appointment appt)
{
  int i = 0;
  while (i < apptList.size())
  {
    if (appt.conflictsWith((Appointment)(apptList.get(i))))
    {
      apptList.remove(i);
    }
    else
    {
      i++;
    }
  }
}
```

ALTERNATE SOLUTION

```
public void clearConflicts(Appointment appt)
{
  for (int i = apptList.size()-1; i >= 0; i--)
  {
    if (appt.conflictsWith((Appointment)apptList.get(i)))
    {
      apptList.remove(i);
    }
  }
}
```

Question 1: Daily Schedule (continued)

PART C:

```
public boolean addAppt(Appointment appt, boolean emergency)
{
  if (emergency)
  {
    clearConflicts(appt);
  }
  else
  {
    for (int i = 0; i < apptList.size(); i++)
    {
      if (appt.conflictsWith((Appointment)apptList.get(i)))
      {

        return false;
      }
    }
  }
  return apptList.add(appt);
```

Question 3: Sparse Array

Part (a) `getValueAt` **3 points**

Intent: *Return the value at row index* `row` *and column index* `col` *in sparse array*

+1 Accesses all necessary elements of `entries` (*No bounds errors*)

+1 Identifies element of `entries` at row index `row` and column index `col`, if exists

+1 Returns identified value or returns 0 if no entry exists in `entries` with row index `row` and column index `col`

Part (b) `removeColumn` **6 points**

Intent: *Remove column* `col` *from sparse array*

+1 Decrements `numCols` exactly once

+1 Accesses all elements of `entries` (*No bounds errors*)

+1 Identifies and removes entry with column index `col`

+2 Process entries with column index > `col` within loop

- **+1** Creates new `SparseArrayEntry` with current row index, column index -1, current value
- **+1** Identifies and replaces entry with column index > `col` with created entry

+1 On exit: All and only entries with column index `col` have been removed and all and only entries with column index > `col` have been changed to have column index -1. All other entries are unchanged. (*Minor loop errors ok*)

Question-Specific Penalties

-2 (t) Consistently uses incorrect name instead of `entries`

-1 (u) Directly accesses private instance variables in `SparseArrayEntry` object

2015.3

Sparse Array Part (a):

```
public int getValueAt(int row, int col){
           for (SparseArrayEntry e : entries){
              if (e.getRow() == row && e.getCol() == col){
                 return e.getValue();
              }
           }
           return 0;
    }
```

Part (b):

```
public void removeColumn(int col){
       int i=0;
       while (i < entries.size()){
            SparseArrayEntry e = entries.get(i);
            if (e.getCol() == col){
                entries.remove(i);
            } else if (e.getCol() > col){
                entries.set(i, new SparseArrayEntry(e.getRow(),
                                                    e.getCol()-1,
                                                    e.getValue()));
                i++;
            } else {
                i++;
            }
      }
      numCols--;
}
```

2014.1

Word Scramble

Part (a):

```
public static String scrambleWord(String word){
   int current = 0;
   String result="";
   while (current < word.length()-1){
      if (word.substring(current,current+1).equals("A") &&
            !word.substring(current+1,current+2).equals("A")){
         result += word.substring(current+1,current+2);
         result += "A";
         current += 2;
      }
      else {
         result += word.substring(current,current+1);
         current++;
      }
   }
   if (current < word.length()){
      result += word.substring(current);
   }
   return result;
}
```

Part (b):

```
public static void scrambleOrRemove(List<String> wordList){
   int index = 0;
   while (index < wordList.size()){
      String word=wordList.get(index);
      String scrambled=scrambleWord(word);
      if (word.equals(scrambled)){
         wordList.remove(index);
      }
      else {
         wordList.set(index,scrambled);
         index++;
      }
   }
}
```

2011.3

Fuel Depot

Part (a):

```
public int nextTankToFill(int threshold) {
  int minLevel = this.tanks.get(0).getFuelLevel();
  int minTankIndex = 0;
  for (int i = 1; i < this.tanks.size(); i++) {
    if (this.tanks.get(i).getFuelLevel() < minLevel) {
      minLevel = this.tanks.get(i).getFuelLevel();
      minTankIndex = i;
    }
  }
  if (minLevel <= threshold) {
    return minTankIndex;
  } else {
    return this.filler.getCurrentIndex();
  }
}
```

// Alternative solution

```
public int nextTankToFillA(int threshold) {
  int minTankIndex = this.filler.getCurrentIndex();
  for (int i = 0; i < this.tanks.size(); i++) {
    if (this.tanks.get(i).getFuelLevel() <= threshold &&
        this.tanks.get(i).getFuelLevel() <
          this.tanks.get(minTankIndex).getFuelLevel()) {
      minTankIndex = i;
    }
  }
  return minTankIndex;
}
```

Part (b):

```
public void moveToLocation(int locIndex) {
  if (this.filler.getCurrentIndex() > locIndex) {
    if (this.filler.isFacingRight()) {
      this.filler.changeDirection();
    }
    this.filler.moveForward(this.filler.getCurrentIndex() - locIndex);
  }
  if (this.filler.getCurrentIndex() < locIndex) {
    if (!this.filler.isFacingRight()) {
      this.filler.changeDirection();
    }
    this.filler.moveForward(locIndex - this.filler.getCurrentIndex());
  }
}
```

2008.1

Flight List

PART A:

```
public int getDuration()
{
  if (flights.size() == 0)
  {
    return 0;
  }
  else
  {
    Time start = flights.get(0).getDepartureTime();
    Time end = flights.get(flights.size()-1).getArrivalTime();
    return start.minutesUntil(end);
  }
}
```

PART B:

```
public int getShortestLayover()
{
  if (flights.size() < 2)
  {
    return -1;
  }
  else
  {
    int shortest = getDuration();
    for (int i = 0; i < flights.size()-1; i++)
    {
      Time arrive = flights.get(i).getArrivalTime();
      Time leave = flights.get(i+1).getDepartureTime();
      int layover = arrive.minutesUntil(leave);
      if (layover < shortest)
      {
        shortest = layover;
      }
    }
    return shortest;
  }
}
```

Mutiple Choice Answer

Basic Part – 1 Multiple Choice Answer

1	C
2	D
3	C
4	A
5	E
6	E
7	C
8	E
9	E
10	C
11	C
12	B
13	E
14	E
15	E
16	D
17	A
18	A
19	A
20	C
21	B
22	D
23	E

Basic Part – 2 Multiple Choice Answer

1	D
2	D
3	D
4	D
5	D
6	E
7	C
8	C
9	A
10	C
11	D
12	B

1D Array Multiple Choice Answer

1	E
2	A
3	A
4	C
5	B
6	B
7	E
8	A
9	D
10	C
11	E
12	A
13	C
14	A
15	E

2D Array Multiple Choice Answer

1	D
2	D
3	D
4	E
5	C
6	B
7	E
8	A
9	E

String Multiple Choice Answer

1	D
2	A
3	D
4	B
5	C
6	C
7	A
8	E

Math Multiple Choice Answer

1	C
2	E
3	E
4	D
5	E
6	E

Class Multiple Choice Answer

1	E
2	B
3	D
4	B
5	D
6	A
7	B
8	D
9	C
10	B
11	D

ArrayList Multiple Choice Answer

1	B
2	E
3	D
4	C
5	E
6	B
7	B
8	B
9	D
10	C

Inheritance Multiple Choice Answer

1	D
2	B
3	B
4	A
5	B
6	B

Recursion Multiple Choice Answer

1	B
2	E
3	D
4	B
5	B
6	D
7	C
8	A
9	D
10	A
11	C

Sorting and Searching Multiple Choice Answer

1	B
2	B
3	C
4	D
5	C

Other Answer

1	E