FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

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

Notes:

- 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.

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 $\neq$ 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.

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

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

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.

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.

    }
```

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 $n\text{th}$ 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,

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

`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)
```

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

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

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.

This question involves the use of *check digits*, which can be used to help detect if an error has occurred when a number is entered or transmitted electronically. An algorithm for computing a check digit, based on the digits of a number, is provided in part (a).

The `CheckDigit` class is shown below. You will write two methods of the `CheckDigit` class.

```
public class CheckDigit
{
    /** Returns the check digit for num, as described in part (a).
      * Precondition: The number of digits in num is between one and six,
     inclusive.
      * num >= 0
      */
    public static int getCheck(int num)
    {
        /* to be implemented in part (a) */
    }

    /** Returns true if numWithCheckDigit is valid, or false otherwise, as
    described in part (b).
     * Precondition: The number of digits in numWithCheckDigit is between two
    and seven, inclusive.
      * numWithCheckDigit >= 0
     */
    public static boolean isValid(int numWithCheckDigit)
    {
        /* to be implemented in part (b) */
    }

    /** Returns the number of digits in num. */
    public static int getNumberOfDigits(int num)
    {
        /* implementation not shown */
    }

    /** Returns the nth digit of num.
      * Precondition: n >= 1 and n <= the number of digits in num
      */
    public static int getDigit(int num, int n)
    {
        /* implementation not shown */
    }

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

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

3. **(a)** Write the `getCheck` method, which computes the check digit for a number according to the following rules.

Multiply the first digit by 7, the second digit (if one exists) by 6, the third digit (if one exists) by 5, and so on. The length of the method's `int` parameter is at most six; therefore, the last digit of a six-digit number will be multiplied by 2.
Add the products calculated in the previous step.
Extract the check digit, which is the rightmost digit of the sum calculated in the previous step.

The following are examples of the check-digit calculation.

Example 1, where num has the value 283415

The sum to calculate is
$(2x7) + (8x6) + (3x5) + (4x4) + (1x3) + (5x2) = 14 + 48 + 15 + 16 + 3 + 10 = 106$.
The check digit is the rightmost digit of 106, or 6, and `getCheck` returns the integer value `6`.

Example 2, where num has the value 2183

The sum to calculate is $(2x7) + (1x6) + (8x5) + (3x4) = 14 + 6 + 40 + 12 = 72$.
The check digit is the rightmost digit of 72, or 2, and `getCheck` returns the integer value `2`.

Two helper methods, `getNumberOfDigits` and `getDigit`, have been provided.

`getNumberOfDigits` returns the number of digits in its `int` parameter.
`getDigit` returns the `nth` digit of its `int` parameter.

The following are examples of the use of `getNumberOfDigits` and `getDigit`.

Method Call	**Return Value**	**Explanation**
`getNumberOfDigits(283415)`	`6`	The number `283415` has 6 digits.
`getDigit(283415, 1)`	`2`	The first digit of `283415` is `2`.
`getDigit(283415, 5)`	`1`	The fifth digit of `283415` is `1`.

Complete the `getCheck` method below. You must use `getNumberOfDigits` and `getDigit` appropriately to receive full credit.

```
/** Returns the check digit for num, as described in part (a).
 * Precondition: The number of digits in num is between one and six,
inclusive.
 * num >= 0
 */
public static int getCheck(int num)
```

(b) Write the `isValid` method. The method returns `true` if its parameter `numWithCheckDigit`, which represents a number containing a check digit, is valid, and `false` otherwise. The check digit is always the rightmost digit of `numWithCheckDigit`.

The following table shows some examples of the use of `isValid`.

Method Call	**Return Value**	**Explanation**
`getCheck(159)`	`2`	The check digit for `159` is `2`.
`isValid(1592)`	`true`	The number `1592` is a valid combination of a number (`159`) and its check digit (`2`).
`isValid(1593)`	`false`	The number `1593` is not a valid combination of a number (`159`) and its check digit (`3`) because `2` is the check digit for `159`.

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

```
/** Returns true if numWithCheckDigit is valid, or false otherwise, as
described in part (b).
 * Precondition: The number of digits in numWithCheckDigit is between two
 and seven, inclusive.
 * numWithCheckDigit >= 0
 */
public static boolean isValid(int numWithCheckDigit)
```

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

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

Notes:

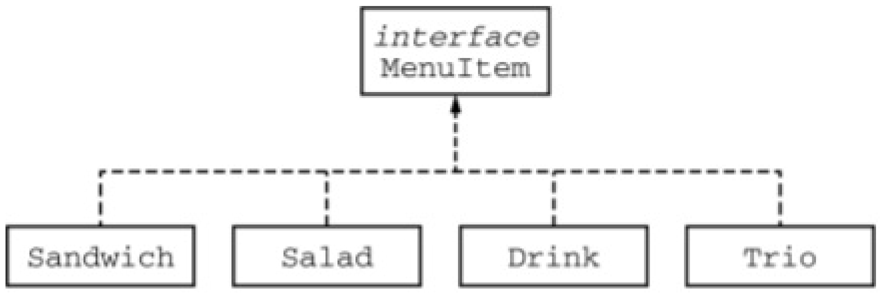
- Unless otherwise noted in the question, assume that parameters in method calls are not null and that methods are called only when their preconditions are satisfied.
- In writing solutions, you may use any of the accessible methods that are listed in classes defined in the question. Writing significant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

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

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

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

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

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



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

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

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

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

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

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

```
Sandwich sandwich;
Salad salad;
Drink drink;
/* Code that initializes sandwich, salad, and drink */

Trio trio = new Trio(sandwich, salad, drink);   // Compiles without error

Trio trio1 = new Trio(salad, sandwich, drink); // Compile-time error
Trio trio2 = new Trio(sandwich, salad, salad); // Compile-time error
```

Write the complete Trio class below.

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

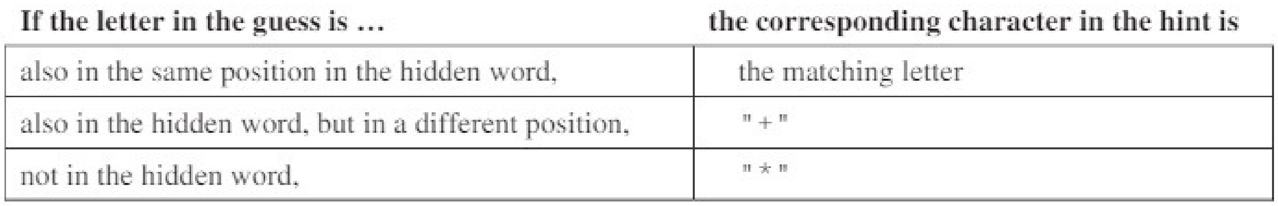
5. **Directions:** SHOW ALL YOUR WORK. REMEMBER THAT PROGRAM SEGMENTS ARE TO BE WRITTEN IN JAVA.

Notes:

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

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.



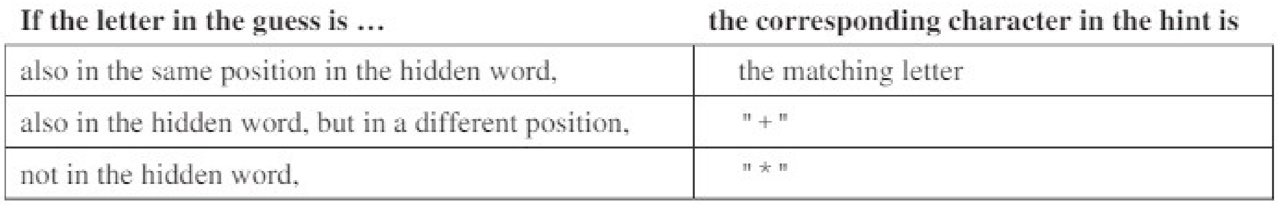
If the letter in the guess is ...	the corresponding character in the hint is
also in the same position in the hidden word,	the matching letter
also in the hidden word, but in a different position,	"+"
not in the hidden word,	"*"

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.



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,	"*"

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

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.

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

6. 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`

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

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)
```

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

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

- 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.

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.

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

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`.

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

Write the complete `GameSpinner` class. Your implementation must meet all specifications and conform to the example.

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

8. In this question, you will complete methods in classes that can be used to represent a multi-player game. You will be able to implement these methods without knowing the specific game or the players' strategies.

The GameState interface describes the current state of the game. Different implementations of the interface can be used to play different games. For example, the state of a checkers game would include the positions of all the pieces on the board and which player should make the next move.

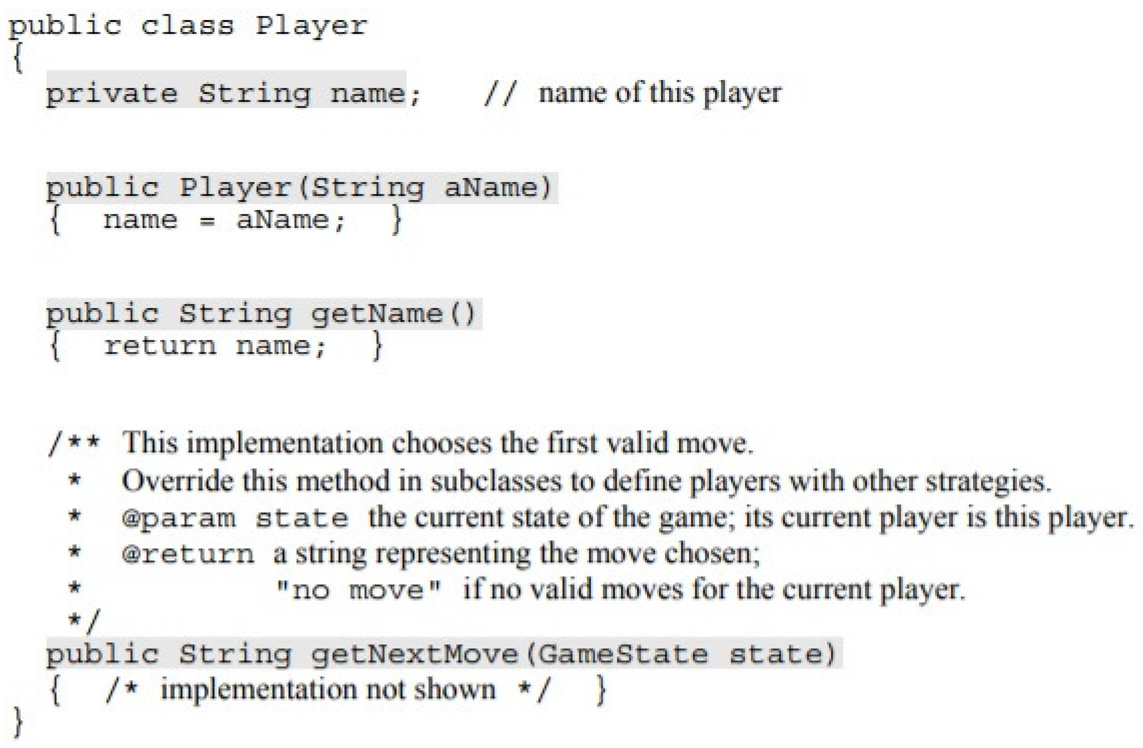
The GameState interface specifies these methods. The Player class will be described in part (a).

```
public interface GameState
{
   /** @return true if the game is in an ending state;
    *          false otherwise
    */
   boolean isGameOver();


   /** Precondition: isGameOver() returns true
    *  @return the player that won the game or null if there was no winner
    */
   Player getWinner();


   /** Precondition: isGameOver() returns false
    *  @return the player who is to make the next move
    */
   Player getCurrentPlayer();


   /** @return a list of valid moves for the current player;
    *          the size of the returned list is 0 if there are no valid moves.
    */
   ArrayList<String> getCurrentMoves();


   /** Updates game state to reflect the effect of the specified move.
    *  @param move a description of the move to be made
    */
   void makeMove(String move);


   /** @return a string representing the current GameState
    */
   String toString();
}
```

The makeMove method makes the move specified, updating the state of the game being played. Its parameter is a String that describes the move. The format of the string depends on the game. In tic-tac-toe, for example, the move might be something like "X-1-1", indicating an X is put in the position (1, 1).

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

a. The Player class provides a method for selecting the next move. By extending this class, different playing strategies can be modeled.



```
public class Player
{
  private String name;    // name of this player

  public Player(String aName)
  {  name = aName;  }

  public String getName()
  {  return name;  }

  /** This implementation chooses the first valid move.
   *  Override this method in subclasses to define players with other strategies.
   *  @param state the current state of the game; its current player is this player.
   *  @return a string representing the move chosen;
   *          "no move" if no valid moves for the current player.
   */
  public String getNextMove(GameState state)
  {  /* implementation not shown */  }
}
```

The method getNextMove returns the next move to be made as a string, using the same format as that used by makeMove in GameState. Depending on how the getNextMove method is implemented, a player can exhibit different game-playing strategies.

Write the complete class declaration for a RandomPlayer class that is a subclass of Player. The class should have a constructor whose String parameter is the player's name. It should override the getNextMove method to randomly select one of the valid moves in the given game state. If there are no valid moves available for the player, the string "no move" should be returned.

b. The GameDriver class is used to manage the state of the game during game play. The GameDriver class can be written without knowing details about the game being played

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

```
public class GameDriver
{
  private GameState state;   // the current state of the game

  public GameDriver(GameState initial)
  {  state = initial;  }


  /** Plays an entire game, as described in the problem description
   */
  public void play()
  {   /*  to be implemented in part (b)  */  }

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

Write the GameDriver method play. This method should first print the initial state of the game. It should then repeatedly determine the current player and that player's next move, print both the player's name and the chosen move, and make the move. When the game is over, it should stop making moves and print either the name of the winner and the word "wins" or the message "Game ends in a draw" if there is no winner. You may assume that the GameState makeMove method has been implemented so that it will properly handle any move description returned by the Player getNextMove method, including the string "no move".

Complete method play below

```
/** Plays an entire game, as described in the problem description
 */
public void play()
```

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

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

- 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.

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)
```

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

```
    { /* 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).

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

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)
```

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)
```

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

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

- 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.

A manufacturer wants to keep track of the average of the ratings that have been submitted for an item using a running average. The algorithm for calculating a running average differs from the standard algorithm for calculating an average, as described in part (a).

A partial declaration of the `RunningAverage` class is shown below. You will write two methods of the `RunningAverage` class.

```
public class RunningAverage
{
     /** The number of ratings included in the running average. */
     private int count;

     /** The average of the ratings that have been entered. */
     private double average;

     // There are no other instance variables.

     /** Creates a RunningAverage object.
       * Postcondition: count is initialized to 0 and average is
       * initialized to 0.0.
       */
     public RunningAverage()
     { /* implementation not shown */ }

     /** Updates the running average to reflect the entry of a new
       * rating, as described in part (a).
       */
     public void updateAverage(double newVal)
     { /* to be implemented in part (a) */ }

     /** Processes num new ratings by considering them for inclusion
       * in the running average and updating the running average as
       * necessary. Returns an integer that represents the number of
       * invalid ratings, as described in part (b).
       * Precondition: num > 0
       */
```

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

```
        public int processNewRatings(int num)
        { /* to be implemented in part (b) */ }

        /** Returns a single numeric rating. */
        public double getNewRating()
        { /* implementation not shown */ }
    }
```

(a) Write the method `updateAverage`, which updates the `RunningAverage` object to include a new rating. To update a running average, add the new rating to a calculated total, which is the number of ratings times the current running average. Divide the new total by the incremented count to obtain the new running average.

For example, if there are 4 ratings with a current running average of 3.5, the calculated total is 4 times 3.5, or 14.0. When a fifth rating with a value of 6.0 is included, the new total becomes 20.0. The new running average is 20.0 divided by 5, or 4.0.

Complete method `updateAverage`.

```
/** Updates the running average to reflect the entry of a new
 * rating, as described in part (a).
 */
public void updateAverage(double newVal)
```

(b) Write the `processNewRatings` method, which considers `num` new ratings for inclusion in the running average. A helper method, `getNewRating`, which returns a single rating, has been provided for you.

The running average must only be updated with ratings that are greater than or equal to zero. Ratings that are less than 0 are considered invalid and are not included in the running average.

The `processNewRatings` method returns the number of invalid ratings. See the table below for three examples of how calls to `processNewRatings` should work.

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

Statement	Ratings Generated	`processNewRatings` Return Value	Comments
`processNewRatings(2)`	`2.5, 4.5`	`0`	Both new ratings are included in the running average.
`processNewRatings(1)`	`-2.0`	`1`	No new ratings are included in the running average.
`processNewRatings(4)`	`0.0,` `-2.2,` `3.5,` `-1.5`	`2`	Two new ratings (`0.0` and `3.5`) are included in the running average.

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

```
/** Processes num new ratings by considering them for inclusion
 * in the running average and updating the running average as
 * necessary. Returns an integer that represents the number of
 * invalid ratings, as described in part (b).
 * Precondition: num > 0
 */
public int processNewRatings(int num)
```

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

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

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

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

The following examples illustrate the behavior of CodeWordChecker objects.

Example 1

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

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

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

Example 2

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

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

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

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

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

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

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

- 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.

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

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

`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.

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

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

- 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.

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");
```

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

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)
```

(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");
```

FRQ / Unit 1 and 2 FRQ: 1 Methods and Control Structures 2 Class

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)
```