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**

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.*

2019

AP CollegeBoard

AP® Computer Science A

Scoring Guidelines

2019 SCORING GUIDELINES

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., 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.*

2019 SCORING GUIDELINES

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`.

2019 SCORING GUIDELINES

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

2019 SCORING GUIDELINES

Question 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;
}
```

These canonical solutions serve an expository role, depicting general approaches to solution. Each reflects only one instance from the infinite set of valid solutions. The solutions are presented in a coding style chosen to enhance readability and facilitate understanding.

2019 SCORING GUIDELINES

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

2019 SCORING GUIDELINES

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

2019 SCORING GUIDELINES

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 SCORING GUIDELINES

Question 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;
      }
   }
}
```

These canonical solutions serve an expository role, depicting general approaches to solution. Each reflects only one instance from the infinite set of valid solutions. The solutions are presented in a coding style chosen to enhance readability and facilitate understanding.

2019 SCORING GUIDELINES

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

2019 SCORING GUIDELINES

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 SCORING GUIDELINES

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;
   }
}
```

These canonical solutions serve an expository role, depicting general approaches to solution. Each reflects only one instance from the infinite set of valid solutions. The solutions are presented in a coding style chosen to enhance readability and facilitate understanding.

2019 SCORING GUIDELINES

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

2019 SCORING GUIDELINES

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 SCORING GUIDELINES

Question 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];
}
```

These canonical solutions serve an expository role, depicting general approaches to solution. Each reflects only one instance from the infinite set of valid solutions. The solutions are presented in a coding style chosen to enhance readability and facilitate understanding.

AP® Computer Science A 2016 Scoring Guidelines

2016 GENERAL SCORING GUIDELINES

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.*

2016 SCORING GUIDELINES

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 CANONICAL SOLUTIONS

Question 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));
}
```


2016 SCORING GUIDELINES

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 CANONICAL SOLUTIONS

Question 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;
}
```


2016 SCORING GUIDELINES

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 CANONICAL SOLUTIONS

Question 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);
                        }
                  }
            }
      }
}
```


2016 SCORING GUIDELINES

Question 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*)

2016 CANONICAL SOLUTIONS

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;
}
```