2. This question involves methods that distribute text across lines of an electronic sign. The electronic sign and the text to be displayed on it are represented by the `Sign` class. You will write the complete `Sign` class, which contains a constructor and two methods.

   The `Sign` class constructor has two parameters. The first parameter is a `String` that contains the message to be displayed on the sign. The second parameter is an `int` that contains the *width* of each line of the sign. The width is the positive maximum number of characters that can be displayed on a single line of the sign.

   A sign contains as many lines as are necessary to display the entire message. The message is split among the lines of the sign without regard to spaces or punctuation. Only the last line of the sign may contain fewer characters than the width indicated by the constructor parameter.

   The following are examples of a message displayed on signs of different widths. Assume that in each example, the sign is declared with the width specified in the first column of the table and with the message `"Everything on sale, please come in"`, which contains 34 characters.

Width of the Sign	Sign Display

15	`Everything on s`
	`ale, please com`
	`e in`

17	`Everything on sal` `e, please come in`
40	`Everything on sale, please come in`

   In addition to the constructor, the `Sign` class contains two methods.

   The `numberOfLines` method returns an `int` representing the number of lines needed to display the text on the sign. In the previous examples, `numberOfLines` would return `3`, `2`, and `1`, respectively, for the sign widths shown in the table.

   The `getLines` method returns a `String` containing the message broken into lines separated by semicolons `(;)` or returns `null` if the message is the empty string. The constructor parameter that contains the message to be displayed will not include any semicolons. As an example, in the first row of the preceding table, `getLines` would return `"Everything on s;ale, please com;e in"`. No semicolon should appear at the end of the `String` returned by `getLines`.

GO ON TO THE NEXT PAGE.

The following table contains a sample code execution sequence and the corresponding results. The code execution sequence appears in a class other than `Sign`.

Statement	Method Call Return Value (blank if none)	Explanation
`String str;`		
`int x;`		
`Sign sign1 = new Sign("ABC222DE", 3);`		The message for `sign1` contains 8 characters, and the sign has lines of width 3.
`x = sign1.numberOfLines();`	`3`	The sign needs three lines to display the 8-character message on a sign with lines of width 3.
`str = sign1.getLines();`	`"ABC;222;DE"`	Semicolons separate the text displayed on the first, second, and third lines of the sign.
`str = sign1.getLines();`	`"ABC;222;DE"`	Successive calls to `getLines` return the same value.
`Sign sign2 = new Sign("ABCD", 10);`		The message for `sign2` contains 4 characters, and the sign has lines of width 10.
`x = sign2.numberOfLines();`	`1`	The sign needs one line to display the 4-character message on a sign with lines of width 10.
`str = sign2.getLines();`	`"ABCD"`	No semicolon appears, since the text to be displayed fits on the first line of the sign.
`Sign sign3 = new Sign("ABCDEF", 6);`		The message for `sign3` contains 6 characters, and the sign has lines of width 6.
`x = sign3.numberOfLines();`	`1`	The sign needs one line to display the 6-character message on a sign with lines of width 6.
`str = sign3.getLines();`	`"ABCDEF"`	No semicolon appears, since the text to be displayed fits on the first line of the sign.
`Sign sign4 = new Sign("", 4);`		The message for `sign4` is an empty string.
`x = sign4.numberOfLines();`	`0`	There is no text to display.
`str = sign4.getLines();`	`null`	There is no text to display.
`Sign sign5 = new Sign("AB_CD_EF", 2);`		The message for `sign5` contains 8 characters, and the sign has lines of width 2.
`x = sign5.numberOfLines();`	`4`	The sign needs four lines to display the 8-character message on a sign with lines of width 2.
`str = sign5.getLines();`	`"AB;_C;D_;EF"`	Semicolons separate the text displayed on the four lines of the sign.

Write the complete `Sign` class. Your implementation must meet all specifications and conform to the examples shown in the preceding table.

GO ON TO THE NEXT PAGE.

3. This question involves the analysis of weather data. The following `WeatherData` class has an instance variable, `temperatures`, which contains the daily high temperatures recorded on consecutive days at a particular location. The class also contains methods used to analyze that data. You will write two methods of the `WeatherData` class.

```
public class WeatherData
{
    /** Guaranteed not to be null and to contain only non-null entries */
    private ArrayList<Double> temperatures;

    /**
     *  Cleans the data by removing from temperatures all values that are less than
     *  lower and all values that are greater than upper, as described in part (a)
     */
    public void cleanData(double lower, double upper)
    {  /* to be implemented in part (a) */  }

    /**
     *  Returns the length of the longest heat wave found in temperatures, as described in
     *  part (b)
     *  Precondition: There is at least one heat wave in temperatures based on threshold.
     */
    public int longestHeatWave(double threshold)
    {  /* to be implemented in part (b) */  }

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

GO ON TO THE NEXT PAGE.

Question 2: Class — 9 points

Canonical solution

9 points

```
public class Sign
{
   private String message;
   private int width;

   public Sign(String m, int w)
   {
      message = m;
      width = w;
   }

   public int numberOfLines()
   {
      int len = message.length();
      if (len % width == 0)
      {
         return len / width;
      }
      else
      {
         return len / width + 1;
      }
   }

   public String getLines()
   {
      int linesNeeded = numberOfLines();
      if (linesNeeded  == 0)
      {
         return null;
      }

      String signLines = "";
      for (int i = 1; i < linesNeeded; i++)
      {
         signLines += message.substring((i – 1) * width,
                      i * width) + ";";
      }
      return signLines +
             message.substring((linesNeeded – 1) * width);
   }
}
```

`Sign`

	Scoring Criteria	Decision Rules	
1	Declares class header: `class Sign`	Responses **will not** earn the point if they • declare the class as something other than `public`	**1 point**
2	Declares appropriate `private` instance variable(s) and constructor initializes instance variables using appropriate parameters	Responses **can** still earn the point even if they • store calculated values instead of the message and width, as long as the declared instance variables can collectively answer the questions and their values are computed from the parameters (correctly or incorrectly) Responses **will not** earn the point if they • declare the variable outside the class, or in the class within a method or constructor	**1 point**
3	Declares constructor header: `Sign(String ___, int ___)`	Responses **can** still earn the point even if they • calculate values in the constructor that are returned by other methods, correctly or incorrectly, as long as the parameter types are correct Responses **will not** earn the point if they • declare the constructor as something other than `public`	**1 point**

4	Declares method headers:
	`public int numberOfLines()`
	`public String getLines()`
		Responses **will not** earn the point if they
		• omit either method
		• omit `public` or declare the method as something other than `public`
			1 point

5	`numberOfLines` divides the message length by the line width	Responses **can** still earn the point even if they • perform the division in a method other than `numberOfLines` • perform the division without using the division operator by counting line-width-sized portions of the message or by counting lines produced by the line-delimiting algorithm • incorrectly account for the final line • use a method name inconsistent with the examples, as long as it is recognizably equivalent	**1 point**
6	`numberOfLines` returns appropriate value (*algorithm*)	Responses **can** still earn the point even if they • perform the return value calculation in the constructor • return a different number of lines than `getLines` produces, as long as the number returned is the correct number for the message	**1 point**

	Responses **will not** earn the point if they • end the constructed output with a `;` or extraneous spaces • modify the contents of `message` or `width` after they have been initialized *(no additional –1y penalty)*

Question-specific penalties
None

Total for question 2 9 points

Alternate canonical:

```
public class Sign
{
   private int numLines;
   private String lines;

   public Sign(String msg, int width)
   {
      if (!msg.equals(""))
      {
         lines = "";
         while (msg.length() > width)
         {
            lines += msg.substring(0, width) + ";";
            msg = msg.substring(width);
            numLines++;
         }
         lines += msg;
         numLines++;
      }
   }

   public int numberOfLines()
   {
      return numLines;
   }

   public String getLines()
   {
      return lines;
   }
}
```

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.*