FRQ 1 — Methods & Control Structures

The StringAnalyzer class stores a string and provides methods to analyze it.

```
public class StringAnalyzer
{
    private String str;

    public StringAnalyzer(String s)
    {
        str = s;
    }

    public int countVowels()
    {
        /* part (a) */
    }

    public boolean hasMoreVowels()
    {
        /* part (b) */
    }
}
```

Part (a): `countVowels()`

- We create a variable `count` to keep track of how many vowels we find.
- We loop through the string one character at a time using a `for` loop.
- For each character:
    - Check if it is one of: `'a', 'e', 'i', 'o', 'u'`.
- If it is a vowel, we increment `count`.
- At the end, we return the total.

public int countVowels()
{

Part (b): `hasMoreVowels()`

- We reuse our method from part (a): `countVowels()`
    - This is important — it avoids duplicating logic.
- We compare:

```
None
```

- `number of vowels > half the string length`

-

- If true → return `true`, otherwise `false`

public boolean hasMoreVowels()
{

Warmup: `Assignment` Class

A class `Assignment` represents a student assignment. Each assignment when created will store a title and the maximum possible score. A method (`recordScore`) will allow the user to add the student's score (which should not be greater than the maximum score).

For example, the following code:

```
None
Assignment hw = new Assignment("Homework 1", 50);

hw.recordScore(40);

System.out.println(hw.getPercent());
System.out.println(hw.isPassing());
```

could print:

```
None
0.8
true
```

Your Task

Write the complete `Assignment` class.

Your class should:

- Have a constructor that takes in title and maximum score
- Allow a score to be recorded for the assignment
- Return the student's percentage as a decimal
- Determine whether the assignment is passing (at least 70%)

Notes

- A new assignment should start with a student score of `0`.
- `getPercent()` should return the percent as a decimal.
    - For example, `40 / 50` should return `0.8`.
- If `recordScore` is given a score higher than the maximum score, store the maximum score instead.

ShoppingCart FRQ Warmup

The `Item` class represents a product with a name and a price.

```
None
public class Item
{
   private String name;
   private int price;

   public Item(String n, int p)
   {
       name = n;
       price = p;
   }

   public String getName()
   {
       return name;
   }

   public int getPrice()
   {
       return price;
   }
}
```

A `ShoppingCart` stores a list of `Item` objects.

```
None
import java.util.ArrayList;

public class ShoppingCart
{
   private ArrayList<Item> items;

   public ShoppingCart(ArrayList<Item> list)
   {
       items = list;
   }
```

```
 // returns the number of items with price less than maxPrice
   public int countCheapItems(int maxPrice)
   {









   }

   // removes all items with price greater than maxPrice
   public void removeExpensiveItems(int maxPrice)
   {









   }

   // returns the name of the item with the greatest price
   public String getMostExpensiveName()
   {









   }
}
```

Exceeds

public void sortByPrice() //sorts the items in the cart from cheapest to most expensive

SeatingChart FRQ Warmup

A `SeatingChart` represents a classroom seating arrangement using a 2D array of student names.

Each position in the array represents one seat in the classroom.

The string `"empty"` represents an unoccupied seat.

None

```
public class SeatingChart
{
   private String[][] seats;

   public SeatingChart(String[][] s)
   {
       seats = s;
   }

   // returns the number of empty seats
   public int countEmptySeats()
   {
       /* to be implemented in part (a) */
   }

   // returns true if the given student
   // appears in the chart and has an "empty" seat directly next to them horizontally.
   public boolean hasOpenSeatNextTo(String name)
   {
       /* to be implemented in part (b) */
   }

   // EXCEEDS moves all "empty" seats
   // to the last column of each row
   public void moveEmptySeats()
   {
      /* to be implemented in part (c) */
   }



}
```

(a) Write the `countEmptySeats` method.

The method should return the number of seats in the chart that contain the string `"empty"`.

For example, if `seats` contains:

```
{
   {"Ava", "empty", "Liam"},
   {"Noah", "Emma", "empty"}
}
```

then the call:

```
countEmptySeats()
```

returns:

```
2
```

because there are two seats labeled `"empty"`.

(b)

Example:

```
{
   {"Ava", "empty", "Liam"},
   {"Noah", "Emma", "Sophia"}
}
```

```
hasOpenSeatNextTo("Ava")
```

returns `true`.

Write the `moveEmptySeats` method.

For each row in the seating chart, the method should move all occurrences of `"empty"` to the end of the row while keeping the order of the other student names the same.

For example, if `seats` originally contains:

```
None
{
   {"Ava", "empty", "Liam"},
   {"empty", "Emma", "Noah"}
}
```

after the call:

```
None
moveEmptySeats();
```

the array should contain:

```
None
{
   {"Ava", "Liam", "empty"},
   {"Emma", "Noah", "empty"}
}
```

because all occupied seats shift left while `"empty"` seats move to the end of each row.

```java
for (int row = 0; row < grid.length; row++) {
    for (int col = 0; col < grid[row].length; col++) {
        System.out.println(
            "Value at [" + row + "][" + col + "] = " + grid[row][col]
        );
    }
}
```

```
for (int r=0;r<seats.length;r++){
      for (int c=0;c<seats[r].length;c++){



      }
}
```