CIT 594 Module 7 Programming Assignment
CSV Reader

In this assignment you will read files in a format known as "comma separated values" (CSV) and output the data represented by the file.

Learning Objectives

In completing this assignment, you will:

- Implement a method to extract data from an input CSV
- Read and understand a formal specification for the CSV format
- Use a "state machine"

Background and Getting Started

Applications have long used delimited text files to store and transmit tables. The simplicity of the format means these files are human readable and editable as well as relatively easy to support in code. The breadth of support from nearly any general purpose application or tabular data loading library continues to make these files particularly popular for publishing data despite their limitations and inefficiencies. You will see more of this in later assignments where you will be required to support reading publicly available datasets.

One of the more common choices for delimiters are commas to separate fields within a row of the table, and line breaks to separate rows of the table. The common name for these comma based file formats are "comma separated values" (CSV).

To support values in the fields that include delimiter characters (e.g., if a comma is part of the data, not just a field separator) requires some added complexity. There is no single, universally-accepted specification for CSV files, so we will focus on one specific format that is widely accepted for this assignment, RFC 4180.

To get started, carefully read sections 1 and 2 of RFC 4180:

https://datatracker.ietf.org/doc/html/rfc4180

That document uses a precise syntax to express an exact formal specification. Read the relevant portion of linked supporting documents to make sure you properly interpret that syntax.

CSV Format For This Assignment

You will write a method to process CSV files character-by-character. The exact format is a relaxation of RFC 4180. There are minor changes to two rules:

```
CRLF = [CR] LF
TEXTDATA =  %x00-09 / %x0B-0C / %x0E-21 / %x23-2B / %x2D-7F
```

The first rule change makes the carriage return (CR) optional everywhere CRLF is used. This is a relaxation seen in many places to adjust for the fact that many systems and applications choose to omit the carriage return and only use a single line feed character for line breaks.

The second rule change expands TEXTDATA[1] to all characters that are not comma, double quote, carriage return, and line feed.

Note: these are relaxations, therefore all documents that are valid for the strict interpretation of RFC 4180 are valid for this format.

For the purposes of the assignment there are five classes of characters for you to consider:

[1]We will not test with characters above $\%x7F$ (127), but you are welcome to include $\%xFF - 7FFFFFFF$ (Integer.MAX_VALUE) in TEXTDATA if you wish. That will cover many more special characters and allow you to process Unicode values.

Common Name	RFC Name	RFC Code (hex)	Decimal	Java Character
Line Feed	LF	`%x0A`	10	`\n`
Carriage Return	CR	`%x0D`	13	`\r`
Double Quote	DQUOTE	`%x22`	34	`"`
Comma	COMMA	`%x2C`	44	`,`
Anything Else	TEXTDATA			`[^\r\n,"]`[2]

Activity

Implement the `readRow` method in `CSVReader.java`.

You may also add supporting fields and helper methods to the `CSVReader` object as needed. Each call to `readRow` must return one row of data from `reader` until the input is exhausted.

If there is a format error in the input, you should raise a `CSVFormatException`. The `CSVFormatException` constructors require 4 integers that are used to indicate two different representations of the position of the error: the [line, character] numbers of the error location with respect to the input stream and the [record, field] numbers of the error location with respect to the logical structure of the table represented by the CSV file. All four values should be 1-indexed. The end of file marker should be treated as just another character in terms of position.

For example, if the input is entirely empty, the result should be `CSVFormatException(1,1,1,1)`. You may add messages if you like — these may be helpful for your debugging and testing and will be ignored by the grader.

Once no further data is available (whether due to a `CSVFormatException` or due to getting to the end of the input) `readRow` should return `null`.

Performance matters. The overall runtime for reading through the entire input should be **O(n)** where **n** is the number of characters in the input. As a reminder, this does mean that certain seemingly convenient operations and data structures may not be appropriate for this assignment. Choose your data structures carefully.

You are encouraged to process the input one character at a time (hence the requirement to use our provided **CharacterReader** rather than the standard **java.io.Reader**). Along those lines, you may wish to organize your code in a "state machine". The term "state machine" just implies that a program may respond differently to a given input based on a current "state" or context. A more detailed explanation is included with the assignment as supplemental reading.

[2] This is just a regular expression to write any other character aside from the four listed.

Before You Submit

Please be sure that:

- your classes are in the default package, i.e. there is no "package" declaration at the top of the source code
- your classes compile and you have not changed the signature of the methods we have provided
- you have not created any additional **.java** files
- you have not overloaded any existing method names
- you have filled out the required academic integrity signature in the comment block at the top of your submission files

How to Submit

After you have finished implementing the *CSVReader* class, go to the "Module 7 Programming Assignment submission" item and click the "Open Tool" button to go to the Codio platform.

Once you are logged into Codio, read the submission instructions in the `README` file. Be sure you upload your code to the "submit" folder.

To test your code before submitting, click the "Run Test Cases" button in the Codio toolbar.

Unlike the Module 1 Programming Assignment, **this will run some but not all of the tests that are used to grade this assignment.** That is, there **are** "hidden tests" on this assignment!

The test cases we provide here are "sanity check" tests to make sure that you have the basic functionality working correctly, but **it is up to you to ensure that your code satisfies all of the requirements described in this document.** Just because your code passes all the tests when you click "Run Test Cases", that doesn't mean you'll get 100% when you submit that code for grading!

When you click "Run Test Cases," you'll see quite a bit of output, even if all tests pass, but at the bottom of the output you will see the number of successful test cases and the number of failed test cases.

You can see the name and error messages of any failing test cases by scrolling up a little to the "Failures" section.

Assessment

Your code will be evaluated against test cases that stress the limits of the specification including tricky but valid inputs and invalid inputs. There is no ambiguity in the specification. If something does not fit the grammar it is not valid.

Grading will be roughly:

- 0% trivial CSV inputs
- 20% simple valid CSV inputs
- 60% tricky valid inputs
- 20% invalid inputs

As noted above, the tests that are executed when you click "Run Test Cases" are not all of the tests that are used for grading. There are "hidden" tests for each of the three methods described here.

After submitting your code for grading, you can go back to this assignment in Codio and view the "results.txt" file, which should be listed in the Filetree on the left. This file will describe any failing test cases.

FAQ and Hints

- "What about..." → RFC 4180.
- **java.lang.StringBuilder**
- **java.io.Reader**
- Characters in C and Java are written with single quotes (`'a'`). Double quotes are for strings (`"this is a string"`).
- The `switch` statement is your friend (at least for this assignment).

Processing trivial CSV files (e.g., a file with quotes) can be as simple as:

```
Files.lines(Path.of(filename)).map(line -> line.split(",")).toArray()
```

This might help you get started with some initial testing. This is also just the baseline functionality for your implementation, you will receive no credit if your implementation does not correctly handle more sophisticated tests.