csv4j User Guide

Table of contents

1. Introduction

csv4j provides an easy to use Open Source API for reading and writing Comma Separated Values (CSV) files. The API distinguishes itself by providing a higher level processing API on top of the low-level parsing layer.

csv4j fully supports the CSV format as defined in RFC4180 which includes values containing embedded carriage returns and line feeds as well as commas. This format can be read and written by Microsoft Excel. The API supports the optional header and allows the developer to make use of it in its use of interfaces such as `Map`. In addition, support is provided to allow delimiters other than the comma (,). Hence it really supports (DSV) delimiter separated value format.

For further reading on the CSV format:

- Comma Separated Values on Wikipedia
- RFC 4180
- How To: The Comma Separated Value (CSV) File Format

2. Overview

The API provides several classes for reading and parsing CSV input:

- **`CSVParser`** which parses lines of CSV text into a List<String>.
- **`CSVTokenizer`** which provies an API similar to `StringTokenizer`.
- **`CSVReader`** which reads from a `Reader` source and parses CSV lines.

To make *processing* of CSV input simple, the API provides a `CSVFileProcessor` class which will iterate thru the file, parse each line, and call the user-supplied processor to actually use the parsed line of text. This allows the application developer to focus on writing code that works with the list of CSV values and not worry about parsing CSV text. The API provides several views of the parsed data:

1. Standard list view: values are provided as a List<String>
2. Map view: values are provided in Map<String,String> where the key is column name from the CSV header line and the value is the column value for a particular line. The map can either be sorted (according to the order from the header line) or unsorted (for faster performance).

In addition, support for writing CSV is provided by the **`CSVWriter`** class.

3. Getting Started

csv4j requires JDK 1.5 or higher. The only jars required is the csv4j.jar provided in the distribution.

3.1. Parsing a line of text using the CSVTokenizer

The standard Java library provides the StringTokenizer class which takes a string and tokenizes it based on the specified delimiter. This class however does not support true CSV format. The csv4j API provides a class with a similar interface called `CSVTokenizer`. It supports a variety of interfaces making it easy to use as we demonstrate in the source example below.

Example using CSVTokenizer like a StringTokenizer:

```
import net.sf.csv4j.CSVTokenizer;

final CSVTokenizer tokenizer = new CSVTokenizer( "a, b ,\" c1,c2,c3 \"," );
while ( tokenizer.hasMoreTokens() ) {
    final String token = tokenizer.nextToken();
    System.out.println( String.format( "token='%s'", token ) );
}
```

The CSV text consists of 4 tokens:

1. 'a'
2. ' b '
3. ' c1,c2,c3 '
4. '' (empty string)

Note that the tokens contain any whitespace; to automatically trim the tokens use the non-default constructor and specify *trimFields* as `true`.

Example using CSVTokenizer like an Iterator:

```
import net.sf.csv4j.CSVTokenizer;

final CSVTokenizer tokenizer = new CSVTokenizer( "a,b,\" c1,c2,c3 \"," );
for ( final String token : tokenizer ) {
    System.out.println( String.format( "token='%s'", token ) );
}
```

3.2. Parsing a line of text using the CSVParser

The `CSVParser` provides an interface which can be used to repeatedly parse CSV text. Unlike the tokenizer interface, it returns the complete set of tokens as a `List` of `String` objects.

Example using CSVParser:

```
import java.util.List;
import net.sf.csv4j.CSVParser;

final CSVParser parser = new CSVParser();
final List<String> tokens = parser.tokenize( "a,b,\" c1,c2,c3  \"," );
for ( final String token : tokens ) {
    System.out.println( String.format( "token='%s'", token ) );
}
```

3.3. Processing a line of text

The example below shows how to use the `CSVFileProcessor` class to process a CSV text file. The `processFile` method takes a `CSVLineProcessor` parameter which allows the user to process the header line (typically the first line) and one or more data lines as each line is being parsed.

The `continueProcessing()` method allows the user to signal that the file processing should stop. This is most useful when processing a large file and an the application no longer needs to continue reading the file.

Example using CSVFileProcessor with a CSVLineProcessor:

```
import java.util.List;
import net.sf.csv4j.CSVFileProcessor;
import net.sf.csv4j.CSVLineProcessor;


final CSVFileProcessor fp = new CSVFileProcessor();
fp.processFile( "datafile.csv", new CSVLineProcessor() {

    public void processHeaderLine( final int linenumber, final List<String>
fieldNames )
    {
        // do something with the list of header columns
        for ( final String fieldName : fieldNames ) {
            System.out.println( fieldName );
        }
    }


    public void processDataLine( final int linenumber, final List<String>
fieldValues )
    {
        // use the data
        for ( final String fieldValue : fieldValues ) {
            System.out.println( fieldValue );
        }
    }
} );
```

Instead of viewing the CSV file as lines containing lists of Strings, the API provides a Map view which takes advantage of the CSV header line which contains the names of each of the columns. The Map view then returns a Map for every line consisting of field name-value pairs.

The Map can be either unsorted or sorted. The `SortedMap` maintains the order of the fields in the line. If ordering is not important, the unsorted `Map` should be used since it has better performance.

Example usage of a CSVFileProcessor with a (unsorted) CSVFieldMapProcessor:

```
import java.util.Map;
import net.sf.csv4j.CSVFileProcessor;
import net.sf.csv4j.CSVFieldMapProcessor;

final CSVFileProcessor fp = new CSVFileProcessor();
fp.processFile( "datafile.csv", new CSVFieldMapProcessor() {

    public void processDataLine( final int linenumber, final
Map<String,String> fields ) {
        // print out the field names and values
        for ( final Entry field : fields.entrySet() ) {
            System.out.println( String.format( "Line #%d: field: %s=%s",
linenumber, field.key(), field.value() );
        }
    }

    public boolean continueProcessing()
    {
        return true;
    }
} );
```

3.4. Writing CSV output

The `CSVWriter` class provides methods to write CSV output on a line-by-line basis. Output is written to the specified Writer. The CSVWriter supports writing CSV lines as `List` of `String` objects using the `writeLine` or specific types of lines

1. Comment lines via `writeCommentLine()`
2. Header line via `writeHeaderLine()`
3. Data lines via `writeDataLine()`

Example usage of a CSVWriter:

```
import java.io.FileWriter;
import net.sf.csv4j.CSVWriter;
```

```
final FileWriter fileWriter = new FileWriter( "output1.csv" );
final CSVWriter csvWriter = new CSVWriter( fileWriter );
csvWriter.writeCommentLine( "This is a sample CSV output file generated by
csv4j's CSVWriter");
csvWriter.writeLine( new String[] { "column1", "column2", "column3" } );
for ( int ii = 0; ii < 10; ii++ ) {
    csvWriter.writeLine( new String[] { "abc", "def,123",
String.format("%d", ii ) } );
}
fileWriter.close();
```

The generated file contains:

```
# This is a sample CSV output file generated by csv4j's CSVWriter
column1,column2,column3
abc,"def,123",0
abc,"def,123",1
abc,"def,123",2
abc,"def,123",3
abc,"def,123",4
abc,"def,123",5
abc,"def,123",6
abc,"def,123",7
abc,"def,123",8
abc,"def,123",9
```

Note that the column2 values require double quotes since it contains an embedded comma (,).