



FlatPack for Java 3.1.0

This document should be used in conjunction with the samples and Java Docs, which come with the distribution.

<i>History</i>	<i>2</i>
<i>Installation</i>	<i>3</i>
<i>Working with Delimited Files.....</i>	<i>4</i>
Parse Using XML Map.....	4
Factory Method	4
Parse Using Database Table Layout	5
Factory Method	5
Parse Using Existing Column Names in File	6
Factory Method	6
Handling Delimiters and Qualifiers Inside Of Data Elements	7
Handling Line Breaks In Delimited Files	8
<i>Working With Fixed Length Files</i>	<i>9</i>
Parse Using Database Table Layout	9
Constructor(s).....	9
Parse Using PZ Map XML	10
Factory Method	10
<i>Retrieving Parsing Errors.....</i>	<i>11</i>
Adding Custom Errors.....	11
<i>Sorting Data</i>	<i>12</i>
<i>Header And Trailer Records.....</i>	<i>13</i>
<i>Exporting To Excel.....</i>	<i>14</i>
Filtering Columns On Export.....	14
<i>Replacing Data.....</i>	<i>15</i>
<i>Exception Handling / Logging</i>	<i>16</i>
<i>SLF4J</i>	<i>16</i>
<i>Read Files With Minimal Memory Usage.....</i>	<i>17</i>
<i>3.0 To 3.1 Migration Notes</i>	<i>18</i>

History

The base code for FlatPack was started from a project I worked on at my job. At the time, I was writing quite a few file imports which were mostly fixed width. I kept encountering the same problem; we had to add something to the file layout somewhere, or expand a length, thus changing all the substrings in the code. I decided that there must be a way to map out the file so that changing the file layout would not break the code. This is when FlatPack was born, although it did not have a name as of yet. The first iteration of the code had the field mappings in a database table, and seemed to work very well for my projects at work.

At that time, I had been spending a lot of time on the Java Sun forums. The same questions kept re-appearing. How do I read a CSV file, or how do I read fixed text. I decided that with a little more work, my project could benefit the community. Whenever I had some free time at home I started to make enhancements to the code. I developed a way to map columns with an XML file instead of having to store the mapping in a database, and a generic parser to handle any kind of delimited file, the delimited and qualifier were passed into the constructor.

This brings us to today. Since the first release, there have been many fixes / enhancements to the parser, mainly the delimited parser. My hope is that this project will take off and become a fixture in the community. If you have a good experience with this project, and it has benefited you in some way, please spread the word.

Recently, [ObjectLab](#) from the UK has decided to offer some support to FlatPack. FlatPack is now "kind of" part of the ObjectLab Kit family, a 'support' group for useful open source projects. They developed the Maven build, the website and are active members of this project. They are world leaders in the design and development of bespoke applications for the Securities Finance Industry.

As of 05/2007 the PZFileReader project was renamed to the project now known as FlatPack. This name provides a better description of the API, which serves as a toolbox for flat files.



Installation

JDOM (www.jdom.org) is a **required dependency** of FlatPack. JDOM is used to parse the pzmap files. The “jdom.jar” which is packaged in the lib folder of the distro for your convenience, and the “FlatPackX_X_X.jar” must be on your class path. If mapping out the columns in the file through a database table, the driver for your database must also be on the class path.

SLF4J (<http://www.slf4j.org>) is a **required dependency**. SLF4J is used to control the logging of events, which may occur during a parsing operation. SLF4J is a simple facade for logging systems allowing the end-user to plug-in the desired logging system at deployment time. If no logger is specified, logging will be done in the console.

JExcelApi (<http://sourceforge.net/projects/jexcelapi>) is an optional jar. This is used to export DataSets to Excel. The “jxl.jar” is packaged in the lib folder of the distro for your convenience.

Working with Delimited Files

The FlatPack parser will handle any type of delimited file, with or without text qualifiers. These include, but are not limited to; CSV, tab, semicolon, etc, just to name a few. FlatPack also allows for a mix of qualified and unqualified elements within a record:

```

"a","b","c"
"a",b,c
a,b,c
"a (" qualifier and (,) delimiter in string","b","c"

```

The above examples are all valid.

Parse Using XML Map

Column names are mapped to fields in the file through an XML document. The fields specified in the XML document must be in the same order as they appear in the text file. This methodology is recommended if the column names are not provided in the first line of the file, or you would like to use different column names than what is coming across in the file.

Example (see Delimited.pzmap.xml

<http://flatpack.sourceforge.net/documentation/index.html> for full example):

```

<PZMAP>
  <COLUMN name="FIRSTNAME" />
  <COLUMN name="LASTNAME" />
</PZMAP>

```

Factory Method

```

DefaultParserFactory.getInstance().newDelimitedParser
(java.io.Reader pzmapXML,
 java.io.Reader dataSource,
 char delimiter,
 char qualifier,
 boolean ignoreFirstRecord)

```

pzmapXML: File object pointing to the XML mapping file.

dataSource: File object pointing to the text file to be parsed.

delimiter: Character indicating how the file is delimited
(comma, tab, semicolon, etc.)

qualifier: Character indicating what is qualifying the text in the file. If this is not applicable, pass FPConstants.NO_QUALIFIER

ignoreFirstRecord: Boolean, when true, indicates the first record in the file contains the column names and should be skipped.

Parse Using Database Table Layout

Column names are mapped to fields in the file through tables in a database. The fields specified in the table must be in the same order as they appear in the text file. This methodology is recommended if the column names are not provided in the first line of the file, or you would like to use different column names than what is coming across in the file.

See `SQLTableLayout.txt` in the references folder for an explanation on the table structure needed:

Factory Method

```
DefaultPZParserFactory.getInstance().newDelimitedParser  
(java.sql.Connection con,  
    java.io.Reader dataSource,  
    java.lang.String dataDefinition,  
    char delimiter,  
    char qualifier,  
    boolean ignoreFirstRecord)
```

con: Connection object to the database which contains the “datafile” and “datastructure” tables.

dataSource: File object pointing to the text file to be parsed.

dataDefinition: String name of the data definition from the “datafile” table.

delimiter: String indicating how the file is delimited (comma, tab, semicolon, etc.)

qualifier: Character indicating what is qualifying the text in the file. If this is not applicable, pass `FPConstants.NO_QUALIFIER`

ignoreFirstRecord: Boolean, when true, indicates the first record in the file contains the column names and should be skipped.

Parse Using Existing Column Names in File

This can be used to retrieve the data out of the rows using the column names which have already been provided in the file. If the column names are provided in the file, but you wish to change the names by which you access the columns, see one of the mapping methodologies above.

Factory Method

```
DefaultPZParserFactory.getInstance().newDelimitedParser  
(java.io.Reader dataSource,  
    char delimiter,  
    char qualifier)
```

dataSource: File object pointing to the text file to be parsed.

delimiter: Character indicating how the file is delimited
(comma, tab, semicolon, etc.)

qualifier: Character indicating what is qualifying the text in the
file. If this is not applicable, pass `FPConstants.NO_QUALIFIER`

Handling Delimiters and Qualifiers Inside Of Data Elements

The FlatPack parser was designed to automatically handle delimiters or qualifiers found within the text of a column. The text must be qualified in order for this functionality to work. Below are some examples of what the parse will handle. The examples assume comma as the delimiter and double quotes for the text qualifier. However, this functionality will work for any delimiter & qualifier combo.

“Here , Is “Some” Text” – **Legal Data Element**

“Here Is”,Some Text” – **Illegal Data Element**

Having a qualifier immediately followed by a delimiter inside the element will break the parse. This is the only situation which must be avoided.

Handling Line Breaks In Delimited Files

Since version 2.1.0 FlatPack has been designed to automatically deal with delimited files, which contain line breaks.

```
ie.  element, element, "element with line break
      more element data
      more element data
      more element data"
      start next rec here
```

The data element containing the line breaks **MUST** be qualified. It does not matter which character is used for the qualifier or the delimiter. These are specified on the constructor prior to the parse.

Working With Fixed Length Files

The FlatPack parser will handle any size fixed length file. There are no limitations on the row byte length. Column positions can be mapped through either the pzmap XML, or a database table. The layouts are described in more detail below.

Parse Using Database Table Layout

Column names are mapped to fields in the file through tables in a database. The fields specified in the table must be in the same order as they appear in the text file, and the length column must be filled in.

See *SQLTableLayout.txt* at <http://flatpack.sourceforge.net/documentation/index.html>

Constructor(s)

```
DefaultPZParserFactory.getInstance().newFixedLenghPaser  
(java.sql.Connection con,  
 java.io.Reader,  
 java.lang.String dataDefinition)
```

con: Connection object to the database which contains the “datafile” and “datastructure” tables. Driver MUST support prepared statements.

dataSource: File object pointing to the text file to be parsed.

dataDefinition: String name of the data definition from the “datafile” table.

Parse Using PZ Map XML

Column names are mapped to fields in the file through an XML document. The fields specified in the XML document must be in the same order as they appear in the text file. The length attribute is mandatory for fixed length files.

Example (see `FixedLength.pzmap.xml` at <http://flatpack.sourceforge.net/documentation/index.html>):

```
<PZMAP>
  <COLUMN name="FIRSTNAME" length="35" />
  <COLUMN name="LASTNAME" length="35" />
</PZMAP>
```

Factory Method

```
DefaultPZParserFactory.getInstance().newFixedLenghPaser  
    (java.io.Reader pzmapXML,  
     java.io.Reader dataSource)
```

pzmapXML: Object pointing to the XML mapping file.
dataSource: Object pointing to the text file to be parsed.

Retrieving Parsing Errors

As the text file is read in and parsed the parser may encounter errors on certain rows of the file. As these errors happen, the errors are thrown into a `DataError` object and added to the errors collection in the `DataSet`. The `DataError` holds 3 pieces of information; the line number in the text file, the error level, and the description of the error. Rows with a `DataError` error level of 1 are warnings. These rows are still included in the `DataSet`, but may require extra attention. Rows with an error level > 1 are not included in the `DataSet`.

The `getErrors()` method will return a collection of `DataError` objects which happened during processing. **This should be checked on every parse.**

Adding Custom Errors

While looping through the data in the file, it is possible to add your own errors to the error collection. These can then be stored up and reported on after `DataSet` processing is completed.

Rows can be added to the error collection by calling the following method in the `DataSet`:

```
public void addError(java.lang.String errorDesc,  
                    int lineNo,  
                    int errorLevel)
```

Sorting Data

FlatPack allows for the DataSet to be resorted by any column(s) and in ASC or DESC order. The ASC / DESC order is specified on a column by column basis, so there can be 2 different sorts in different directions at the same time. After the sort is completed, the DataSet positions the pointer before the first row.

1. Create a new OrderBy object.
2. Add OrderColumns to the OrderBy object. These need to be added in the order in which you would like the sort to take place. Below is a quick example:

```
orderby = new OrderBy();  
orderby.addOrderColumn(new OrderColumn("CITY",false));  
orderby.addOrderColumn(new OrderColumn("LASTNAME",true));  
ds.orderRows(orderby);
```

The example above sorts the DataSet by 2 columns. The primary sort is on the CITY column in ASC order, and the secondary sort is on the LASTNAME column in DESC order.

Header And Trailer Records

As of version 2.2 FlatPack allows for the mapping of header, trailer, or other records, which can be uniquely identified by a piece of data on the record. These must be defined using the PZ Map XML. Here are some example setups:

1. Couple of things to note
 - a. You can have an unlimited number of <record> elements. The order in which they are defined in the xml file is the order, which it will compare against the data. The most common <record> elements should be at the top of your file as this will lead to faster parsing. Columns defined outside of the <record> elements are considered the detail of the file. These mappings will be used if none of the record elements match. *It is still possible to define a mapping without <record> elements if needed.*
2. Delimited File:
 - a. <http://flatpack.sourceforge.net/documentation/index.html> see DelimitedWithHeaderAndTrailer.pzmap.xml.
 - b. Required <record> attributes
 - i. Id – The unique id to give to this header record. IsRecordID() method can then be invoked to see if the DataSet row being processed belongs to this particular id.
 1. example – `if (dataSetVar.IsRecordID("header")) //header record logic`
 - ii. elementNumber – The column number in the file which contains the record indicator. This is 1 based.
 - iii. Indicator – Value, which must be contained in the column for this record mapping to be used.
3. Fixed Width File:
 - a. <http://flatpack.sourceforge.net/documentation/index.html> see: FixedLengthWithHeaderAndTrailer.pzmap.xml
 - b. Required <record> attributes
 - i. Id – The unique id to give to this header record. IsRecordID() method can then be invoked to see if the DataSet row being processed belongs to this particular id.
 1. example – `if (dataSetVar.IsRecordID("header")) //header record logic`
 - ii. startPosition – The starting position of the indicator column. This is 1 based.
 - iii. EndPosition – The end position of the indicator column. This is 1 based.
 - iv. Indicator – Value which must be contained in the column for this record mapping to be used.

Since version 2.1.0, FlatPack allows the DataSet object to be exported to an Excel. This can be done by using the ExcelTransformer() class.

Construct a ExcelTransformer(DataSet, File) object passing the DataSet to write to Excel and a file object of where the Excel file should be written to.

Call writeExcelFile() which will perform the write of the Excel file to the hard disk.

WARNING.... If the specified Excel file already exists, the file **WILL BE** overwritten. Non-existing files will be created automatically.

Filtering Columns On Export

Columns can be excluded / included in a couple different ways. The first way is to tell the class that only certain columns should be written to the Excel sheet. The **setExportOnlyColumns(String[])** will be able to accomplish this. The second way is to tell the class to export all columns except the ones provided. The **setExcludeFromExportColumns(String[])** will provide this functionality. The export only columns take priority over the exclude list if they are both set.

Note:

This functionality makes use of the JexcelApi library, which can be found at:
<http://sourceforge.net/projects/jexcelapi>

This library has been packaged with the download in the lib folder (jxl.jar). This is an optional jar and is only required when converting files to Excel.

Replacing Data

FlatPack allows for the data from the file to be manipulated. These changes are STRICTLY done in memory and make no modifications to the original file. This functionality is done through the setValue(String columnName, String newValue) method in the DataSet. When executed, this change only takes place on the row that the pointer is currently sitting on.

Note:

If changing the value of columns and utilizing the export to Excel functionality, the changes made in the DataSet WILL BE reflected in the Excel document created.

Exception Handling / Logging

Below is a list of common exceptions that may happen when constructing a DataSet:

1. FileNotFoundException “DATA DEFINITION CAN NOT BE FOUND IN THE DATABASE”
 - a. This only applies to the Database Table Map. This would indicate that the datasourceName passed into the constructor does not match a name on the DATAFILE table.
2. FileNotFoundException “pzmap XML file does not exist”
 - a. This only applies to the Pzmap XML file. This would indicate that the pzmapXML File object is null, or pointing to a non-existent file.
3. FileNotFoundException "DATASOURCE DOES NOT EXIST, OR IS NULL. BAD FILE PATH.”
 - a. The text file specified to read does not exist, or is NULL.

SLF4J

Slf4j is utilized to capture logging information that occurred during a parse. By default, this information will be logged to the console. SLF supports the following loggers; log4j, jcl, nop, and jdk1.4. Please go to <http://www.slf4j.org> to download the proper jar for your logging preference. Here is a typical SLF4j setup:

SLF-api-XXX.jar + SLF-MyLoggerPreference.jar + MyLogger.jar (lo4j.jar for example)

Here are the steps that would need to be taken for log4j:

1. Install log4j.jar
2. Setup log4j.properties or log4j.xml
3. Install SLF-api-XXX.jar
4. Install SLF-log4j12-XXX.jar

Read Files With Minimal Memory Usage

If looking to keep the memory overhead very low, then please use the `BufReaderParseFactory` class. These parsers will leave the Reader open while looping through the file. However, there is a great amount of functionality lost with this methodology, and it is recommended that you use the `DefaultParseFactory` if memory is not an issue. This factory is the replacement for `LargeDataSet` in previous versions of `FlatPack`.

IMPORTANT:

In order to release the lock on the file you must do the following at the end of the parse.

1. Cast the Parser received from the factory to `BufReaderFixedParser` / `BufReaderDelimPZParser` depending on which one was called from the factory
2. Call the `close()` method on the parser.
 - a. *If the close method is not called FlatPack will attempt to clean up the open file handles when the parser gets GC'd.*

This will make the file available to the system and release the lock.

THE FOLLOWING METHODS ARE DISABLED FOR THIS FACTORY:

`previous()`, `orderRows()`, `absolute()`, `remove()`, `getIndex()`, `goBottom()`, `goTop()`, `setValue()`

3.0 To 3.1 Migration Notes

See changes document at: <http://flatpack.sourceforge.net/changes-report.html> for full release notes.

1. Change imports from net.sf.pzfilereader to net.sf.flatpack
2. Removed the “PZ” acronym from class names. A full list of the affected class can be found in the changes link above. There were a few classes that had the PZ changed to FP instead of dropping all together. These are; PZConstants, PZConvertException, PZStringUtils
3. It is highly recommended that all uses of java.io.File or java.io.InputStream be changed to an instance of java.io.Reader. This was done in an attempt to make parsing String objects a little cleaner. InputStreams can be changed over to use InputStreamReader and Files can be changed over to FileReader.
4. Removed deprecated versions of the following methods in ParserUtils
 1. isMultiLine()
 2. splitLine()
 3. getCMDKeyForFixedLengthFile()