PZFileReader for Java 2.2.0

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

History

The base code for PZFileReader 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 PZFileReader 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.

Installation

JDOM (www.jdom.org) is a dependency of PZFileReader. 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 “pzFileReaderX_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.

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 PZFileReader 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.

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. 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 in the references folder for full example):

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

Constructor(s)

```
public DataSet(java.io.File pzmapXML,
         java.io.File dataSource,
         java.lang.String delimiter,
         java.lang.String qualifier,
         boolean ignoreFirstRecord,
         boolean handleShortLines)
```

pzmapXML: File object pointing to the XML mapping file.
dataSource: File object pointing to the text file to be parsed.
delimiter: String indicating how the file is delimited (comma, tab, semicolon, etc.)
qualifier: String indicating what is qualifying the text in the file. If this is not applicable, pass a NULL or empty string.
ignoreFirstRecord: Boolean, when true, indicates the first record in the file contains the column names and should be skipped.
handleShortLines: Based on the number of columns in the xml mapping file, the parser expect the specified number of columns to be present on a row. If there are less columns then expected, the row is logged as an error and excluded from the DataSet. When this parameter is set to true, the parser will automatically set the missing columns to empty Strings. Warnings for these rows will be logged in the errors collection.

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:

Constructor(s)

```
public DataSet(java.sql.Connection con,
               java.io.File dataSource,
               java.lang.String dataDefinition,
               java.lang.String delimiter,
               java.lang.String qualifier,
               boolean ignoreFirstRecord,
               boolean handleShortLines)
```

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: String indicating what is qualifying the text in the file. If this is not applicable, pass a NULL or empty string.
ignoreFirstRecord: Boolean, when true, indicates the first record in the file contains the column names and should be skipped.
handleShortLines: Based on the number of columns in the database tabe, the parser expect the specified number of columns to be present on a row. If there are less columns then expected, the row is logged as an error and excluded from the DataSet. When this parameter is set to true, the parser will automatically set the missing columns to empty Strings. Warnings for these rows will be logged in the errors collection.

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. This is new parsing functionality introduced in version 2.0.0.

Constructor(s)

```
public DataSet(java.io.File dataSource,
    java.lang.String delimiter,
   java.lang.String qualifier,
    boolean handleShortLines)
```

dataSource: File object pointing to the text file to be parsed.
delimiter: String indicating how the file is delimited (comma, tab, semicolon, etc.)
qualifier: String indicating what is qualifying the text in the file. If this is not applicable, pass a NULL or empty string.
handleShortLines: Based on the number of columns in the xml mapping file, the parser expect the specified number of columns to be present on a row. If there are less columns then expected, the row is logged as an error and excluded from the DataSet. When this parameter is set to true, the parser will automatically set the missing columns to empty Strings. Warnings for these rows will be logged in the errors collection.

Handling Delimiters and Qualifiers Inside Of Data Elements

The PZFileReader 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 PZFileReader 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 PZFileReader parser will handle any size fixed length file. There are no limitations on the row byte length.

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 in the references folder for an explanation on the table structure needed:

Constructor(s)

public **DataSet**(java.sql.Connection con,
java.io.File dataSource,
java.lang.String dataDefinition,
boolean handleShortLines)

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.
handleShortLines: Based on the total column length in the database table, the parser expects each row to be X number of characters long. If there are less characters then expected, the row is logged as an error and excluded from the DataSet. When this parameter is set to true, the parser will automatically pad the missing characters. Warnings for these rows will be logged in the errors collection.

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 in the references folder for full example):

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

Constructor(s)

```
public DataSet(java.io.File pzmapXML,
               java.io.File dataSource,
               boolean handleShortLines)
```

pzmapXML: File object pointing to the XML mapping file.
dataSource: File object pointing to the text file to be parsed.
handleShortLines: Based on the total column length in the xml file, the parser expects each row to be X number of characters long. If there are less characters then expected, the row is logged as an error and excluded from the DataSet. When this parameter is set to true, the parser will automatically pad the missing characters. Warnings for these rows will be logged in the errors collection.

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

PZFileReader 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 PZFileReader 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 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. In the reference folder 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 in 1 based.
        iii. Indicator – Value which must be contained in the column for this record mapping to be used.
3. Fixed Width File:
    a. In the reference folder 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 in 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.

Exporting To Excel

Since version 2.1.0, pzFileReader allows the DataSet object to be exported to an Excel. This can be done in two ways:

1. Call writeToExcel(File excelFile) in the DataSet class
2. Construct a ExcelTransformer() object and call writeExcelFile()

The first option is going to be the most efficient. If the number of rows in the file exceeds the worksheet limitation for Excel, an exception is thrown.

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

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

PzFileReader 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. 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

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.

Large Text Files

As of PZFileReader 2.2.0, there has been a new class introduced to deal with large files, LargeDataSet. This class extends DataSet, and functionality is virtually the same. There are some methods that have been disabled until I can find a way to implement them in the future:

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

The difference in this parse is the file is no longer read into memory, and the file is left open until the freeMemory() method is called.

There is a /LargeDataSet/LargeCSVPerformanceTest folder which will demonstrate the read of a large file. As the sample calls the next() method on the LargeDataSet, as line will be printed for every 2500 records read.

Deprecated 2.0.0 Constructors

These have all been removed in version 2.1.0