**Epipog**

**Specification**

**Parse Class Family**

**Nov. 16, 2016**

# 1. Base Class

The parse object is the base class for parsing input data that has been read in. Derived objects are extended from the base class, depending on their requirement in the database pipeline. This base class is defined in the file Parse.java. This base class defines the methods and interfaces for how data is parsed in the Epipog application and the operations that may be performed with the parse object.

## 1.1 Fields

The base class defines the following fields:

private String inputFile = null; // File to parse

The path name of the input data file that is being imported.

protected Reader reader = null; // input data reader

The reader object for reading the input data file.

private Boolean hasHeader = true; // input file has header (e.g., csv,psv,tsv)

Flag indicating if input data file contains a header sequence which proceeds the data.

private boolean skip = false; // skip non-parseable input records

Flag indicating whether to skip non-parseable data records. When skipped, parsing proceeds to the next record; otherwise, an exception is thrown and the parsing process is stopped.

private Ejector ejector = new Ejector(); // ejector object for handling non-parseable input

The ejector object for handling non-parsable records. The ejector object, when set, is used to re-route (eject) non-parseable records to another process.

private int nImported = 0; // number of records imported

This field holds the current (accumulator) count of the number of records that have been parsed so far and passed to the import stage (excludes count of records that were ejected).

protected Collection collection = null; // Collection to insert data into

This collection that is bound to the parsed input data (i.e., where the data is inserted into).

## 1.2 Methods

The base class contains the following implemented methods

***Accessors (Getter/Setter)***

public void Header( boolean hasHeader );  
public boolean Header();

The above methods set the flag whether input data file has a proceeding header, and returns the setting of the flag, respectively.

public void Skip( Boolean skip)  
public Boolean Skip();

The above methods set the flag whether to skip non-parseable data records from the input, and returns the setting of the flag, respectively.

public void Reader( Reader.ReaderType type );

This method sets which reader object to instantiate for reading the input data file depending on the setting of the argument ReaderType.

public void Ejector( Ejector.EjectorType type );

This method sets which ejector object to instantiate for handling non-parseable records depending on the setting of the argument EjectorType.

public void Collection( Collection collection );  
public Collection Collection();

The above methods sets the collection object which the imported (parsed) input file is bound to, and returns the collection object, respectively.

***Routing***

protected void Eject( String line );

This method routes and alerts non-parseable records to the ejector object.

public int NEjected();

This getter returns the accumulated number of non-parseable records that have been passed to the ejector object.

public int NImported() ;

This getter returns the accumulated number of records (parseable) that have been passed to the import interface of the bound collection.

***I/O***

public void Open() throws ParseException;

This method invokes the instantiate reader object interface to open the input data file for reading and determine the file size. If the reader object is not instantiated, or file does not exist, or cannot be opened, a ParseException error is thrown.

public void Close();

This method closes the input data file, if it has been opened; otherwise it does nothing. The method does not throw any exception.

## 1.4 Abstract Methods (Interface)

The base class contains abstract methods, which must be implemented by the derived classes, for the following:

***I/O***

public abstract void Parse() throws ParseException;

This method is implemented in the derived classes to handle parsing the input data stream according to its format. If an error occurs, a ParseException is thrown.

***Routing***

protected void Import( Object record );

This method is implemented in the derived classes to handle routing a parsed record to the import interface of a collection that is bound to the input data.

# 2 SVParse Derived Class

This derived class extends base class “Parse”, and implements the methods for parsing the input data file. They are implemented in the file SVParse.java. This extended class implements the methods and interfaces for parsing the input data, when the input is a character separated value file, with one record per line and optionally a heading (i.e., CSV, PSV and TSV).

This derived class performs a type-less parsing of the input data. Only the field names (columns) of the input data need to be known at this stage of the data pipeline, which may either be injected into the parser or derived from the input file data header. The date type of the field does not need to be known, which allows parsing the data and moving it through this part of the pipeline very computationally efficient.

## 2.1 Fields

The extended class defines the following additional fields, which are only accessible (private) by this class.

private char separator = ( char ) 0; // Separator character sequence

This field is the character that separates the fields in a record (e.g., comma in CSV).

private ArrayList<String> heading = null; // column heading (field names and order of columns)

This field holds the heading information as an array list of strings, where the values are the field names (columns) and the array order is the order of the columns in the input data file.

## private int ncols = 0; // number of columns

## The number of expected columns per record in the input data.

## private boolean rfc4180 = false; // enfore RFC 4180 parsing rules

## A flag indicating of the parsing of character separated values strictly according to RFC 4180 MINE type text/csv; otherwise the input data is parsed as a combination of common conventions in the industry.

## 2.2 Methods

The extended class contains the implementation of the following methods:

***Constructor***

public SVParse( String inputFile, char separator );

This constructor takes as arguments the path name of the input file and the character that is used for separating fields (columns). The input file argument is passed to the base class Parse.

***Getters / Setters***

public void Heading( ArrayList<String> heading ) ;

This method is used to pass to the parsing process the expected field names (column) and order. This may be used when the input data does not have a header section, or as a validation step to verify the expected (as defined by this call) and the actual (header in input data) match.

public Integer NCols();

This getter returns the expected number of fields (columns) per record. The number is determined by the expected header or the actual header. Subsequent records with mismatched column count are deemed as non-parseable.

public void RFC4180( boolean rfc4180 );

This setter sets the flag on whether RFC 4180 MIME type text/csv is strictly enforced.

***I/O***

public void Parse() throws ParseException;

This method parses each record in the input data. The method first parses the header, if any, and then sequentially each record. For each record, if the record is non-parseable and the skip flag is set, the non-parseable record is passed to the ejector object and the process continues to the next record. Otherwise, the parsed record is passed to the import object and the process continues to the next record.

This method will throw a ParseException and stop parsing the input data if any of the following occur:

* The reader object is not instantiated.
* There is no expected or actual header information.
* The expected and actual field (column) header count do not match.
* The fields (column) in the actual or expected do not match in name or order.
* A record has the wrong number of columns (fields) and the skip flag is not set.

public static ArrayList<String> Split( String line, char separator, boolean rfc4180, Reader reader )

This method parses a character separated value record (line) and returns the values in the corresponding sequential order. The argument line is the record as a UTF-8 string. The argument separator is the character that is the field separator. The argument rfc4180 is the flag on whether to strictly enforce RFC 4180 parsing rules. The argument reader is the reader object. The reader object is passed for the case where the record is split across more than one line and one or more additional lines need to be read from the input data.

When RFC 4180 is strictly enforced, the record is parsed as follows:

* MS-DOS-style lines that end with (CR/LF) characters (optional for the last line).
* An optional header record (there is no sure way to detect whether it is present, so care is required when importing).
* Each record "should" contain the same number of comma-separated fields.
* Any field may be quoted (with double quotes).
* Fields containing a line-break, double-quote, and/or commas should be quoted. (If they are not, the file will likely be impossible to process correctly).
* A (double) quote character in a field must be represented by two (double) quote characters.

When not enforced, the following additional rules apply:

* fields (columns) are trimmed
* a single quote can appear as a quote if not first character in the line

***Issues***

The following are known issues with the currently implementation:

* enforcing no single double quote outside of double quoted string

***Routing***

protected void Import( Object record );

This method is handles routing a row of parsed data (column-based data) to the import interface of the collection that is bound to the input data.

# 3 CSVParse Derived Class

This derived class extends the class “SVParse”, and implements the methods for parsing the input data file. They are implemented in the file CSVParse.java. This extended class implements the methods and interfaces for parsing the input data, when the input data format is CSV (comma separated values).

## 3.1 Methods

The extended class contains the implementation of the following methods:

***Constructor***

public CSVParse( String inputFile );

This constructor takes as argument the path name of the input. The input file and the character separator comma are passed to the parent class “SVParse”.

# 4 PSVParse Derived Class

This derived class extends the class “SVParse”, and implements the methods for parsing the input data file. They are implemented in the file PSVParse.java. This extended class implements the methods and interfaces for parsing the input data, when the input data format is PSV (pipe separated values).

## 4.1 Methods

The extended class contains the implementation of the following methods:

***Constructor***

public PSVParse( String inputFile );

This constructor takes as argument the path name of the input. The input file and the character separator pipe are passed to the parent class “SVParse”.

# 5 TSVParse Derived Class

This derived class extends the class “SVParse”, and implements the methods for parsing the input data file. They are implemented in the file TSVParse.java. This extended class implements the methods and interfaces for parsing the input data, when the input data format is TSV (tab separated values).

## 5.1 Methods

The extended class contains the implementation of the following methods:

***Constructor***

public TSVParse( String inputFile );

This constructor takes as argument the path name of the input. The input file and the character separator tab are passed to the parent class “SVParse”.

# 6 JSONParse Derived Class

This derived class extends base class “Parse”, and implements the methods for parsing the input data file. They are implemented in the file JSONParse.java. This extended class implements the methods and interfaces for parsing the input data, when the input is in JSON format.

This derived class performs a type-less and schema-less parsing of the input data. The date type of the field does not need to be known, which allows parsing the data and moving it through this part of the pipeline very computationally efficient.

## 6.1 Fields

The extended class defines the following additional fields, which are only accessible (private) by this class.

private boolean bulk = false; // input is single document object or multiple document objects   
  
This field is a flag that tells the parser if the input should be treated as a single document, or a group of documents encapsulated in a root document:

{ “root”: { ... document 1 },  
 { ... document2 },  
 …  
 }

private String token = null; // current token being parsed

This field is an internal variable containing the current token parsed from the input. Tokens are parsed as follows:

Grouping: { } [ ]  
 Separator: , :  
 Strings : “…”  
 Number: numerical

private Integer nObjects = 0; // number of parsed objects

This field is an accumulator that holds the current total number of JSON objects (documents) that have been parsed.  
  
private Integer nFields = 0; // number of parsed fields

This field is an accumulator that holds the current total number of fields (key/value pair) that have been parsed.

## 6.2 Methods

The extended class contains the implementation of the following methods:

***Constructor***

public JSONParse( String inputFile );

This constructor takes as arguments the path name of the input file. The input file argument is passed to the base class Parse.

***Getters / Setters***

public void Bulk( boolean bulk );

This method sets the flag for whether the input data is a single document or a group of documents encapsulated by a single root document.

public Integer NObjects();

This method returns the total number of objects (document) currently parsed.

public Integer NFields();

This method returns the total number of fields (key/value) currently parsed.

public void Parse() throws ParseException

This method parses the JSON input data. The input is processed as a top-level element: object, array, or scalar value. If the top-level element is an object or array, it is further parsed as containing a sequence of fields, where the value component may be either an object, array or scalar value.