CHAPTER 1

NeoCSV

Sven Van Caekenberghe with Damien Cassou and Stéphane Ducasse

CSV (Comma-Separated Values) is a popular data-interchange format. This chapter presents NeoCSV, a library to parse and export CSV files. This chapter has originally been written by Sven Van Caekenberghe the author of NeoCSV and many other nicely designed Pharo libraries.

1.1 NeoCSV

NeoCSV is an elegant and efficient standalone Pharo library to read (resp. write) CSV files converting to (resp. from) Pharo objects.

An Introduction to CSV

CSV is a lightweight text-based *de facto* standard for human-readable tabular data interchange. Essentially, the key characteristics are that CSV (or more generally, delimiter-separated text data):

- is text-based (ASCII, Latin1, Unicode);
- consists of records, 1 per line (any line ending convention);
- where *records* consist of *fields* separated by a *delimiter* (comma, tab, semicolon);
- where every record has the same number of fields; and
- where fields can be quoted should they contain separators or line endings.

Note References: http://en.wikipedia.org/wiki/Comma-separated_values, http://tools.ietf.org/html/rfc4180

Note that there is not one single official standard specification.

Hands On NeoCSV

NeoCSV contains a reader (`NeoCSVReader`) and a writer (`NeoCSVWriter`) to parse and generate delimiter-separated text data to and from Smalltalk objects. The goals of NeoCSV are:

- to be standalone (have no dependencies and little requirements);
- to be small, elegant and understandable;
- to be efficient (both in time and space); and
- to be flexible and non-intrusive.

To load NeoCSV, evaluate the following or use the Configuration Browser:

```
Gofer it
   smalltalkhubUser: 'SvenVanCaekenberghe' project: 'Neo';
   configurationOf: 'NeoCSV';
   loadStable.
```

To use either the reader or the writer, you instantiate them on a character stream and use standard stream access messages.

The first example reads a sequence of data separated by `,` and containing line breaks. The reader produces arrays corresponding to the lines with the data (the `withCRs` method converts backslashes to new lines).

```
(NeoCSVReader on: '1,2,3\4,5,6\7,8,9' withCRs readStream) upToEnd.
   -->  #(#('1' '2' '3') #('4' '5' '6') #('7' '8' '9'))
```

The second proceeds from the inverse: given a set of data as arrays it produces comma separated lines.

```
String streamContents: [ :stream |
  (NeoCSVWriter on: stream)
   nextPutAll: #( (x y z) (10 20 30) (40 50 60) (70 80 90) ) ].
   -->
'"x","y","z"
"10","20","30"
"40","50","60"
"70","80","90"
'
```

1.2 Generic Mode

NeoCSV can operate in generic mode without any further customization. While writing,

- record objects should respond to the `do:` protocol,
- fields are always sent `asString` and quoted, and
- CRLF line ending is used.

While reading,

- records become arrays,
- fields remain strings,
- any line ending is accepted, and
- both quoted and unquoted fields are allowed.

The standard delimiter is a comma character. Quoting is always done using a double quote character. A double quote character inside a field will be escaped by repeating it. Field separators and line endings are allowed inside a quoted field. Any whitespace is significant.

1.3 Customizing NeoCSVWriter

Any character can be used as field separator, for example:

```
neoCSVWriter separator: Character tab
```

or

```
neoCSVWriter separator: $;
```

Likewise, any of the three common line end conventions can be set. In the following example we set carriage return:

```
neoCSVWriter lineEndConvention: #cr
```

There are 3 mechanisms that the writer may use to write a field (in increasing order of efficiency):

quoted converting it with `asString` and quoting it (the default);

raw converting it with `asString` but not quoting it; and

object not quoting it and using `printOn:` directly on the output stream.

When disabling quoting, you have to be sure your values do not contain embedded separators or line endings. If you are writing arrays of numbers for example, this would be the fastest way to do it:

```
neoCSVWriter
   fieldWriter: #object;
   nextPutAll: #( (100 200 300) (400 500 600) (700 800 900) )
```

The `fieldWriter` option applies to all fields.

Writing Objects

If your data is in the form of regular domain-level objects it would be wasteful to convert them to arrays just for writing them as CSV. NeoCSV has a non-intrusive option to map your domain object's fields: You add field specifications based on accessors. This is how you would write an array of `Points`.

```
String streamContents: [ :stream |
   (NeoCSVWriter on: stream)
      nextPut: #('x field' 'y field');
      addFields: #(x y);
      nextPutAll: { 1@2. 3@4. 5@6 } ].
   -->
'"x field","y field"
"1","2"
"3","4"
"5","6"
'
```

Note how `nextPut:` is used to first write the header (*i.e.*, the first line). After printing the header, the writer is customized: the messages `addField:` and `addFields:` arrange for the specified selectors (here `x` and `y`) to be sent on each incoming object to produce fields that will be written.

To change the writing behavior for a specific field, you have to use `addQuotedField:`, `addRawField:` and `addObjectField:`.

To specify different field writers for an array (actually any subclass of `SequenceableCollection`), you can use the `first`, `second`, `third`, etc. methods as selectors:

```
String streamContents: [ :stream |
   (NeoCSVWriter on: stream)
      addFields: #(first third second);
      nextPutAll: { 'acb' . 'dfe' . 'gih' }].
   -->
'"a","b","c"
"d","e","f"
"g","h","i"
'
```

1.4 Customizing NeoCSVReader

The parser is flexible and forgiving. Any line ending will do, quoted and non-quoted fields are allowed.

Any character can be used as field separator, for example:

```
neoCSVReader separator: Character tab
```

or

```
neoCSVReader separator: $;
```

NeoCSVReader will produce records that are instances of its `recordClass`, which defaults to `Array`. All fields are always read as Strings. If you want, you can specify converters for each field, to convert them to integers, floats or any other object. Here is an example:

```
(NeoCSVReader on: '1,2.3,abc,2015/07/07' readStream)
   separator: $,;
   addIntegerField;
   addFloatField;
   addField;
   addFieldConverter: [ :string | Date fromString: string ];
   upToEnd.
   --> an Array(an Array(1 2.3 'abc' 7 July 2015))
```

Here we specify 4 fields: an integer, a float, a string and a date field. Field conversions specified this way only work on indexable record classes, like `Array`.

Ignoring Fields

While reading from a CSV file, you can ignore fields using `addIgnoredField`. In the following example, the third field of each record is ignored:

```
| input |
(NeoCSVReader on: '1,2,a,3\1,2,b,3\1,2,c,3\' withCRs readStream)
   addIntegerField;
   addIntegerField;
   addIgnoredField;
   addIntegerField;
   upToEnd
   -->  #(#(1 2 3) #(1 2 3) #(1 2 3))
```

Adding ignored field(s) requires adding field types on all other fields.

Creating Objects

In many cases you will probably want your data to be returned as one of your domain objects. It would be wasteful to first create arrays and then convert

all those. NeoCSV has non-intrusive options to create instances of your own classes and to convert and set fields on them directly. This is done by specifying accessors and converters. Here is an example for reading `Associations` of `Floats`.

```
(NeoCSVReader on: '1.5,2.2\4.5,6\7.8,9.1' withCRs readStream)
   recordClass: Association;
   addFloatField: #key: ;
   addFloatField: #value: ;
   upToEnd.
   --> {1.5->2.2. 4.5->6. 7.8->9.1}
```

For each field you have to give the mutating accessor to use. You might also want to pass a conversion block using `addField:converter:`.

Reading many Objects

Handling large CSV files is possible with NeoCVS. In the following, we first create a large CSV file then read it partly (this takes a bit of time, be patient).

```
'paul.csv' asFileReference writeStreamDo: [ :file |
   ZnBufferedWriteStream on: file do: [ :out | | writer |
      writer := (NeoCSVWriter on: out).
      writer writeHeader: { #Number. #Color. #Integer. #Boolean}.
      1 to: 1e7 do: [ :each |
         writer nextPut: {
            each.
            #(Red Green Blue) atRandom.
            1e6 atRandom.
            #(true false) atRandom } ] ] ].
```

Above code results in a 300Mb file:

```
$ ls -lah paul.csv
-rw-r--r--@ 1 sven  staff   327M Nov 14 20:45 paul.csv
$ wc paul.csv
10000001 10000001 342781577 paul.csv
```

The following code selectively collects every record with a third field lower than 1000 (this takes a bit of time, be patient):

```
Array streamContents: [ :out |
  'paul.csv' asFileReference readStreamDo: [ :input |
      (NeoCSVReader on: (ZnBufferedReadStream on: in))
         skipHeader;
         addIntegerField;
         addSymbolField;
         addIntegerField;
         addFieldConverter: [ :x | x = #true ];
         do: [ :each |
            each third < 1000
               ifTrue: [ out nextPut: each ] ] ] ].
```