Chapter 1

NeoCSV

CSV (Comma Separated Values) is a popular data-interchange format. NeoCSV is an elegant and efficient standalone Pharo framework to read and write CSV converting to or from Smalltalk objects developed by Sven Van Caekenberghe.

1.1 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
- where fields can be quoted should they contain separators or line endings

Here are some relevant links:

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

Note that there is not one single official standard specification.

1.2 NeoCSV

The goals of NeoCSV are:

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

1.3 Getting Started

NeoCVS is now hosted on SmalltalkHub. To load it you can execute:

```
Gofer new
   smalltalkhubUser: 'SvenVanCaekenberghe' project: 'Neo';
   package: 'ConfigurationOfNeoCSV';
   load

ConfigurationOfNeoCSV project load: #stable
```

The following repository contains the recent code:

```
MCHttpRepository
   location: 'http://smalltalkhub.com/mc/SvenVanCaekenberghe/Neo/main'
   user: ''
   password: ''
```

The NeoCSV framework contains a reader (NeoCSVReader) and a writer (NeoCSVWriter) to parse respectively generate delimiter separated text data to or from Smalltalk objects.

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

Here are two examples:

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

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

1.4 Generic Mode

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

- recorded objects should respond to the `do:` protocol
- fields are always send #asString and quoted
- CRLF line ending is used

While reading,

- records become arrays
- fields remain strings
- any line ending is accepted
- 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. All whitespace is significant.

Note: We should add some examples

1.5 Customizing NeoCSVWriter

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

```
neoCSVWriter separator: Character tab
```

or

```
neoCSVWriter separator: $;
```

Likewise, any of the 3 common line end conventions can be set:

```
neoCSVWriter lineEndConvention: #cr
```

There are 3 ways a field can be written (in increasing order of efficiency):

- quoted - converting it with asString and quoting it (the default)
- raw - converting it with asString but not quoting it
- object - not quoting it and using printOn: directly on the output stream

---

Note: we should add examples

Obviously, 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.

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 can add field specifications based on accessors. For example, this is how you would write an array of Points.

```
neoCSVWriter
  nextPut: #(x y);
  addFields: #(x y);
  nextPutAll: { 1@2. 3@4. 5@6 }
```

Note: what is nextPut: and addFields:

Note how we first write the header (before customizing the writer). The addField: and addFields: messages arrange for the specified accessors to be performed on the incoming objects to produce values that will be written by the fieldWriter. Additionally, there is protocol to specify different field writing behavior per field, using addQuotedField:, addRawField: and addObjectField:.

Note: we should add an example.

To specify different field writers for an array (actually an SequenceableCollection subclass), you can use the methods first, second, third, ... as accessors.

Note: last paragraph is obscure

1.6 Customizing NeoCSVReader

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

Note: add example

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 or floats, any other object. Here is an example:

```
neoCSVReader
  addIntegerField;
  addFloatField;
  addField;
  addFieldConverter: [ :string | Date fromString: string ];
  upToEnd.
```

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.

1.7 Data and Domain objects

In many cases you will probably want your data to be returned as one of your domain objects.

Note: show an example

It would be wasteful to first create arrays and then convert all those. NeoCSV has non-intrusive options to create instances of your own object classes and to convert and set fields on them directly. This is done by specifying accessors and converters.

Examples

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.
```

For each field you give the mutator accessor to use, as well as an implicit or explicit conversion block.

One thing that it will enforce is that all records have an equal number of fields. When there are no field accessors or converters, the field count will be set automatically after the first record is read. If you want you could set it upfront. When there are field accessors or converters, the field count will be set to the number of specified fields.

1.8 Conclusion

NeoCSV is a simple and powerful frameworks to read and emit objects in CSV format. It is customizable to fit your objects and adapt to most situations.