Code examples

Code example: Read a CSV file using its header

To read lines from a file into an object, assume you have the bean class

```
public class UserBean {
    String username, password, street, town;
    int zip;

    public String getPassword() { return password; }
    public String getStreet() { return street; }
    public String getTown() { return town; }
    public String getUsername() { return username; }
    public int getZip() { return zip; }
    public void setPassword(String password) { this.password = password; }
    public void setStreet(String street) { this.street = street; }
    public void setTown(String town) { this.town = town; }
    public void setUsername(String username) { this.username = username; }
    public void setZip(int zip) { this.zip = zip; }
}
```

and that you have a CSV file with a header. Let's assume the following content

```
username, password,   date,        zip,  town
Klaus,    qwexyKiks,  17/1/2007,   1111, New York
Oufu,     bobilop,    10/10/2007,  4555, New York
```

You can then create an instance of the `UserBean` and populate it with values from the second line of the file with the following code

```
class ReadingObjects {
  public static void main(String[] args) throws Exception{
    ICsvBeanReader inFile = new CsvBeanReader(new FileReader("foo.csv"),
CsvPreference.EXCEL_PREFERENCE);
    try {
      final String[] header = inFile.getCSVHeader(true);
      UserBean user;
      while( (user = inFile.read(UserBean.class, header, processors)) !=
null) {
        System.out.println(user.getZip());
      }
    } finally {
      inFile.close();
    }
  }
}
```

What is left to define is `processors`. Despite its condensed form, the cell processors captures quite expressive semantics. Each position in the `CellProcessor` array corresponds to a column in the CSV file. If a cell in the array has the value `null`, the column will be read but not processed. Each

processor can be **nested**. This means that processors and constraints can be freely combined in any number as seen below with the `new Unique(new StrMinMax())`. This is a very expressive so spend some time dwelling on what you can do with the processors. Processors are executed from the outermost to the innermost. In cases of recoverable errors, a processor may choose to exit (optionally with some default value), rather than executing its "nested processors". You can say that while cell processors' constructor is executing from the innermost to the outermost (due to the normal rules in Java on resolving arguments before invoking methods), the processors business logic execute from outside to the innermost.

One definition could be

```
final CellProcessor[] processors = new CellProcessor[] {
    new Unique(new StrMinMax(5, 20)),
    new StrMinMax(8, 35),
    new ParseDate("dd/MM/yyyy"),
    new Optional(new ParseInt()),
    null
};
```

The above definition captures the following semantics:

- Read column 1 as a String and first insure that values in this column are all unique. Then ensure each value has a length between 5-20.
- Read column 2 as a String and ensure each value has a length between 8-35.
- Read column 3 as a date of the format day/month/year.
- Read column 4 as an integer. But only do so if something is actually defined in the column.
- Read column 5 as is (a String). Don't process or constrain its values.

Notice how Super Csv utilizes the fact that the file had a header. If your files are without headers, you can easily create the missing information by means of a string array such as

```
final String[] header = new String[] { "username", "password", "date", "zip",
"town"};
```

If we want to omit a column when reading/writing, simply set the header array element to `null`. This is somewhat analogous to setting a processor array element to `null` which disables processing of that column.

The full code looks as follows

```
import java.io.FileReader;
import java.io.IOException;

import org.supercsv.cellprocessor.Optional;
import org.supercsv.cellprocessor.ParseDate;
import org.supercsv.cellprocessor.ParseInt;
import org.supercsv.cellprocessor.constraint.StrMinMax;
import org.supercsv.cellprocessor.constraint.Unique;
import org.supercsv.cellprocessor.ift.CellProcessor;
import org.supercsv.io.CsvBeanReader;
import org.supercsv.io.ICsvBeanReader;
import org.supercsv.prefs.CsvPreference;

class ReadingObjects {
    static final CellProcessor[] userProcessors = new CellProcessor[] {
        new Unique(new StrMinMax(5, 20)),
```

```
        new StrMinMax(8, 35),
        new ParseDate("dd/MM/yyyy"),
        new Optional(new ParseInt()),
        null
    };

    public static void main(String[] args) throws Exception {
        ICsvBeanReader inFile = new CsvBeanReader(new FileReader("foo.csv"),
CsvPreference.EXCEL_PREFERENCE);
        try {
          final String[] header = inFile.getCSVHeader(true);
          UserBean user;
          while( (user = inFile.read(UserBean.class, header, userProcessors))
!= null) {
            System.out.println(user.getZip());
          }
        } finally {
          inFile.close();
        }
   }
}

public class UserBean {
    String username, password, town;
    Date date;
    int zip;

    public Date getDate() {
        return date;
    }
    public String getPassword() {
        return password;
    }
    public String getTown() {
        return town;
    }
    public String getUsername() {
        return username;
    }
    public int getZip() {
        return zip;
    }
    public void setDate(final Date date) {
        this.date = date;
    }
    public void setPassword(final String password) {
        this.password = password;
    }

    public void setTown(final String town) {
        this.town = town;
    }
    public void setUsername(final String username) {
        this.username = username;
    }
    public void setZip(final int zip) {
        this.zip = zip;
```

```
    }
}
```

Code example: Reading a file with an unknown number of columns.

When you are in the situation, that you know nothing about the structure of the file at compile time, the only option you have is to read each line of the file as a list of Strings. You accomplish this using the `CsvListReader.read()` method.

Code example: Write a file with a header

Writing a file is as easy as reading one. You decide on the column format and start writing. You can still enjoy the cell processor functionality, however, you will most likely only use the constraint parts. Any value is being written to the output stream using the`toString()` method on each object, thus reducing the need for the cell processors that convert data. **Notice that data is automatically "CSV encoded"**. This means that when you write the string `Hello, world`, the actual data written will be `"Hello, world"` as the comma must be escaped since you are only writing one column of data. Similarly, the string `"Hello" world`becomes `"""Hello"" world"` etc. etc. In other words, **You can safely leave it up to Super csv to properly write your data.**

```
import java.util.HashMap;
import org.supercsv.io.*;
import org.supercsv.prefs.CsvPreference;

class WritingMaps {
  main(String[] args) throws Exception {
    ICsvMapWriter writer = new CsvMapWriter(new FileWriter(...),
CsvPreference.EXCEL_PREFERENCE);
    try {
      final String[] header = new String[] { "name", "city", "zip" };
      // set up some data to write
      final HashMap<String, ? super Object> data1 = new HashMap<String,
Object>();
      data1.put(header[0], "Karl");
      data1.put(header[1], "Tent city");
      data1.put(header[2], 5565);
      final HashMap<String, ? super Object> data2 = new HashMap<String,
Object>();
      data2.put(header[0], "Banjo");
      data2.put(header[1], "River side");
      data2.put(header[2], 5551);
      // the actual writing
      writer.writeHeader(header);
      writer.write(data1, header);
      writer.write(data2, header);
```

```
    } finally {
      writer.close();
    }
  }
}
```

Notice, that even though we are writing numbers, we do not need the cell processor, as the numbers `toString()` is called prior to writing. We could have added a cell processor to ensure that the numbers were above a certain threshold, and the string lengths were > 0 if we wanted.

Code examples

Partial reading and writing

Partial reading and writing of objects is not ordinarily a supported feature in CSV frameworks. This short article will present how elegantly partial reading and partial writing is supported in Super CSV. The core of the problem with especially partial writing is that some columns needs be formatted as plain empty if values are missing, whereas other columns, may need to default to particular strings or numbers such as`0` or `-1`.

While it has been suggested that Super CSV's write methods should be extended with a default value in case the field to write is null, the current solution presented below is more flexible and elegantly fits into the existing framework. The solution consists of using the concept of cell processors to convert data (or missing data) to whatever values needed.

All the examples here take outset in a simple order class, where an order may be a partial order of another order (if its `parentOrder` is set to another order).

```
class Order {
    Integer orderNumber;
    Integer parentOrder;
    Integer productNumber;
    String  userComment;

    public Integer getOrderNumber() {
        return orderNumber;
    }
    public void setOrderNumber(int orderNumber) {
        this.orderNumber = orderNumber;
    }
    public Integer getParentOrder() {
        return parentOrder;
    }
    public void setParentOrder(int parentOrder) {
        this.parentOrder = parentOrder;
    }
    public Integer getProductNumber() {
        return productNumber;
    }
    public void setProductNumber(int productNumber) {
        this.productNumber = productNumber;
    }
    public String getUserComment() {
        return userComment;
    }
    public void setUserComment(String userComment) {
```

```
        this.userComment = userComment;
    }
}
```

The class is used in such a fashion that `parentOrder` and `userComment` may only optionally hold values (when appropriate).

Code example: Partially reading objects from a CSV file

This first example shows how to read a partial object from a CSV file utilizing the fact that the CSV file has a header that maps column names to field names of the class.

```
public void should_partial_read() throws Exception {
    // content of a file containing orders and product numbers
    String fileData = "orderNumber, productNumber\n1,22";

    // setup conversion from String to integers (as our fields are type int)
    CellProcessor[] processing = new CellProcessor[] { new ParseInt(), new
ParseInt() };

    // the actual reading
    CsvBeanReader reader = new CsvBeanReader(new StringReader(fileData),
CsvPreference.EXCEL_PREFERENCE);
    // get header to identify what fields to populate
    String[] header = reader.getCSVHeader(true);
    Order order = reader.read(PartialReadWriteExamplesTest.Order.class,
header, processing);

    // show that only part of the object has been populated with values
    System.out.print("order: " + order.getOrderNumber()
            + " product: " + order.getProductNumber()
            + " parent: " + order.getParentOrder() + "\n**\n");
}
```

On the screen is printed

```
order: 1 product: 22 parent: null
**
```

Code example: Writing partial objects to a CSV file

The following example sets up two order objects and writes them to a file. The key to specifying default values to write in case fields are null is `new ConvertNullTo(-1)` and `new ConvertNullTo("\"\"")`. The `null` in the list of processors denotes the columns that need no processing.

```
public void should_partial_write() throws Exception {
    // The data to write
    Order mainOrder = new Order();
    mainOrder.setOrderNumber(1);
    mainOrder.setProductNumber(42);
    mainOrder.setUserComment("some comment");
```

```
    Order subOorder = new Order();
    subOorder.setOrderNumber(2);
    subOorder.setParentOrder(1);
    subOorder.setProductNumber(43);

    // for testing write to a string rather than a file
    StringWriter outFile = new StringWriter();

    // setup header for the file and processors. Notice the match between the
header and the attributes of the
    // objects to write. The rules are that
    // - if optional "parent orders" are absent, write -1
    // - and optional user comments absent are written as ""
    String[] header = new String[] { "orderNumber", "parentOrder",
"productNumber", "userComment" };
    CellProcessor[] Processing = new CellProcessor[] { null, new
ConvertNullTo(-1), null, new ConvertNullTo("\"\"") };

    // write the partial data
    CsvBeanWriter writer = new CsvBeanWriter(outFile,
CsvPreference.EXCEL_PREFERENCE);
    writer.writeHeader(header);
    writer.write(mainOrder, header, Processing);
    writer.write(subOorder, header, Processing);
    writer.close();

    // show output
    System.out.println(outFile.toString());
}
```

On the screen is printed

```
orderNumber,parentOrder,productNumber,userComment
1,-1,42,some comment
2,1,43,""""""
```

Conclusion

That's all there is to it. We have showed how to read and write CSV files partially. We have shown the underlying mechanisms for both reading and writing are the same.

Code examples

Extending SuperCSV with new Cell processors

You can easily extend Super CSV if you need to. Just implement the interface `CellProcessor` and you are set. Most of the existing processors are around 10 lines of code. Let's investigate the implementations by looking at a cell processor which reads a column and converts it into a `Long` object. The cell processors are build around the patters "null object pattern" and "chain of responsibility" hence all the infrastructure is set up outset the processors.

```
 public class ParseLong extends CellProcessorAdaptor {
    /** important to invoke super */
    public ParseLong() {
        super();
    }

    /** important to invoke super */
    public ParseLong(final LongCellProcessor next) {
        super(next);
    }
    /** simplify conversion of column to Long */
    public Object execute(final Object value, final CSVContext context)
throws NumberFormatException {
        final Long result = Long.parseLong((String) value);
        return next.execute(result, context);
    }
}
```

All the magic is taking place in the `execute()` method. It really need no further explaining.

Notice that a processor takes another (possibly chained) processor as an argument. Only if it find its conditions suitable, may it call `next()`. Otherwise it has the option to exit with some error or other value. You can say that while cell processors' constructor is executing from the innermost to the outermost (due to the normal rules in Java on resolving arguments before invoking methods), the processors business logic execute from outside to the innermost.