Simple CSV

This document describes the SimpleCSV library for reading and writing CSV data files. This documentation only briefly describes and explains the library, for more examples look at UnitTest library or in examples. With Simple CSV library for .NET You can write and read CSV documents and serialize/deserialize objects to and from CSV file format. This documentation can be outdated or the source code can contain more functions.

Library version :	1.0.0 RC2
Supported Library :	.NET 3.5
Web :	http://simplecsv.codeplex.com/
Blog :	http://marcind.spaces.live.com
Licence :	Ms-RL

Changes

1.0.0 RC	• Added string indexers in SimpleCSVReader • Added ISimpleCSVSerializationCallback • Added events to serializer (OnSerialization and OnDeserialization) • Added Format property to SimpleCSVAttribute for DateTime
1.0.0 RC2	• Nullable types serialization • Serialization of enumeration types • Line (row) counters in reader and writer • Bug fixes

Reading CSV files

The easiest and simplest way to read an CSV file describes the code below:

```
using (SimpleCSVReader reader = new SimpleCSVReader(@".\BasicReadWrite.csv")) {
  while (reader.ReadLine()) {
    // To read a column value use: reader[i]
  }
}
```

Just access the value by using the indexer ***reader[i]***, where ***i*** is the number of column.

Remember!

Columns in the SimpleCSV are indexed from 1. If the columns does not exists, then *reader[i]* will return a null value. If the column is empty, then *reader[i]* will return an empty string.

If the file contains a header, You can set ***HasHeader*** property to true. But remember, you can change this value only before any read.

```
using (SimpleCSVReader reader = new SimpleCSVReader(@".\BasicReadWrite.csv")) {
  reader.HasHeader = true;
  // Your code
}
```

In this scope, when You access the ***reader.ReadLine()*** method, and a ***HasReader*** property was set to true, then the reader will first read a header line, and then the record. You can manually read a header by executing a ***ReadHeader()*** method before any other read – this does not require to setting the ***HasHeader*** property (it will be set automatically after read)

When You read the file with a header, then You can access the cells values by a header name:

```
using (SimpleCSVReader reader = new SimpleCSVReader(@".\BasicReadWrite.csv")) {
  reader.HasHeader = true;
  while (reader.ReadLine()) {
    // To read a column value use: reader["ColumnName"]
  }
}
```

Remember!

When You access the cell value by header name and without specifying to read the Header before, then an ***InvalidOperationException*** will be thrown.

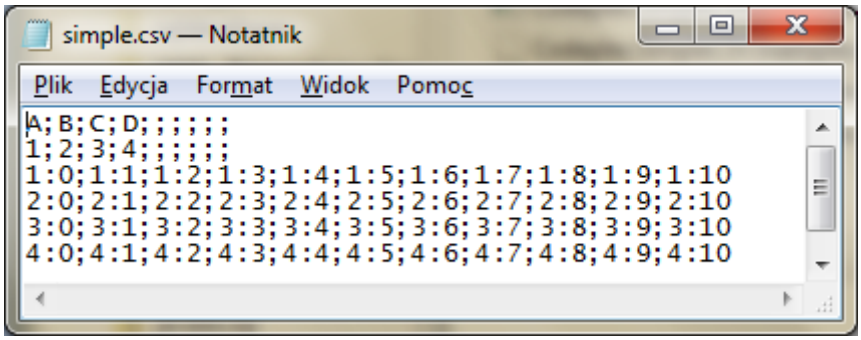
You can change the value of a cell using indexers – this will change only values in memory without modifying the CSV stream.

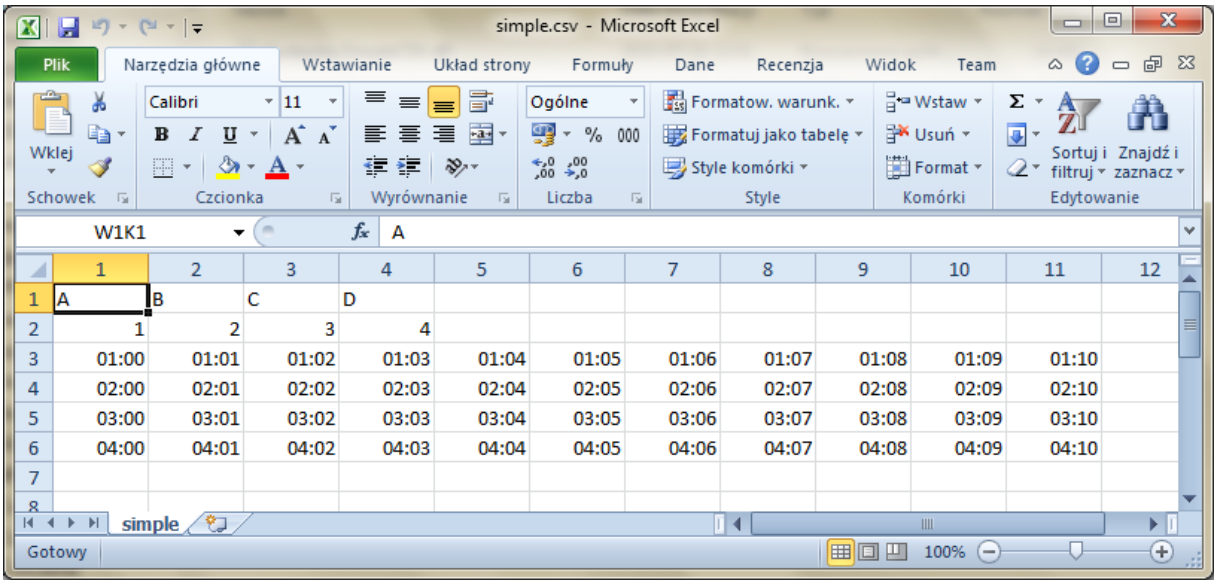
Writing CSV files

Writing CSV files is as simple as reading. You access for example a WriteLine of the writer instance, that's accepts a string array (line 4).

```
01   using (SimpleCSVWriter writer = new SimpleCSVWriter(@".\simple.csv")) {
02     writer.MaxColumns = 10;
03     writer.WriteHeader(new string[] { "A", "B", "C", "D" });
04     writer.WriteLine(new string[] { "1", "2", "3", "4" });
05     for (int i = 1; i < 5; i++) {
06       for (int j = 0; j < 20; j++) {
07          writer.Write(i + ":" + j);
08       }
09       writer.WriteLine();
10     }
11   }
```

On line 3 we are writing a Header to CSV. If You will define, that only 10 columns should be in the exported file, then set ***MaxColumns*** property to the desired value. If You write less or more columns, then only MaxColumns will be saved – when there are less columns saved, then the writer will fill the line with empty columns. Remember, to execute ***WriteLine()*** (with no parameters) when using ***Write*** method. The result (in raw in Notepad and in Microsoft Excell 2010) is presented below (Excell formatted this values automatically)





	1	2	3	4	5	6	7	8	9	10	11	12
1	A	B	C	D								
2	1	2	3	4								
3	01:00	01:01	01:02	01:03	01:04	01:05	01:06	01:07	01:08	01:09	01:10	
4	02:00	02:01	02:02	02:03	02:04	02:05	02:06	02:07	02:08	02:09	02:10	
5	03:00	03:01	03:02	03:03	03:04	03:05	03:06	03:07	03:08	03:09	03:10	
6	04:00	04:01	04:02	04:03	04:04	04:05	04:06	04:07	04:08	04:09	04:10	
7												
8												

You can also use indexers to write data to a cell. Like in reader there are two types of indexer. The first one available all the time is the numeric (integer) indexer, and an string indexer, available only when CSV header is specified.

```
01    for (int i = 1; i < 5; i++)
02    {
03      for (int j = 20; j >= 0; j--)
04      {
05        writer[j] = i + ":" + j;
06      }
07      writer["A"] = "String indexer";
08      writer.WriteLine();
09    }
```

From the previous example we will change only the double loops. The inner loop we will decrease from 20 down to 0, what means, that column 20 will be written first. Outside the loop is used an string indexer with a header label “A” (in the example is the same like numeric indexer with value 1). You can modify indexers till WriteLine is executed.

Remember!

It’s safer to use numeric indexer, because when using string indexers there is a little overhead (with mapping the header to colum), when the header does not match any colum (is not found), then an IndexOutOfRangeException is thrown and what is more, string indexer works only when the header is specified, otherwise an InvalidOperationException will be thrown.

CSV Serialization

For better simplification, there is also an CSV serialization mechanism available. All object properties described with ***SimpleCSVAttribute*** will be handled during serialization. Let's look on a class example, that should be handled with the serializer:

```
01        public class MyObject
02        {
03            [SimpleCSV]
04            public decimal ID { get; set; }
05
06            [SimpleCSV(Label="Name")]
07            public string Title { get; set; }
08
09            [SimpleCSV(Label="Created", Index=4)]
10            public DateTime Date { get; set; }
11
12            [SimpleCSV(Label="Description", Index=3)]
13            public string Data { get; set; }
14
15            public string NotForExport { get; set; }
16        }
```

First property ***ID*** described with ***SilmpleCSV*** attribute will be saved to CSV, and if specified ***HasHeader***, then the column's header name will be named ID. To change the name, just set the value in ***Label***, like in line 6 and 9. What is more, when You will force the position of the columns, then use the ***Index*** value like in lines 9 and 12. Property ***NotForExport*** will not be handlerd during serialization.

Remember!

When the column position defined by Index is not specified, then the column is the position of property in the class described with SimpleCSVAttribute. In the example above, when You change the Index property in line 9, from 4 to 1, then an ***SimpleCSVSerializationException*** will be thrown with the message: *There already is defined a ID column on position 1*.

In the example above, ID and Title will be placed in first and second column, and the Date and Data will be placed in fourth and third column. When You remove the Index properties, then the Date will be placed in third, and Data in fourth column.

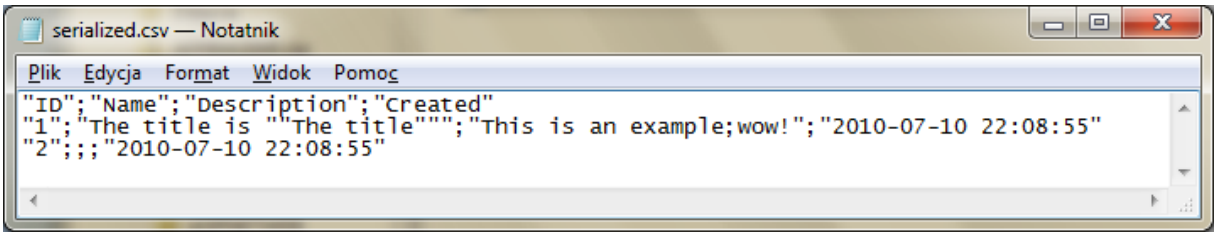
Serializing

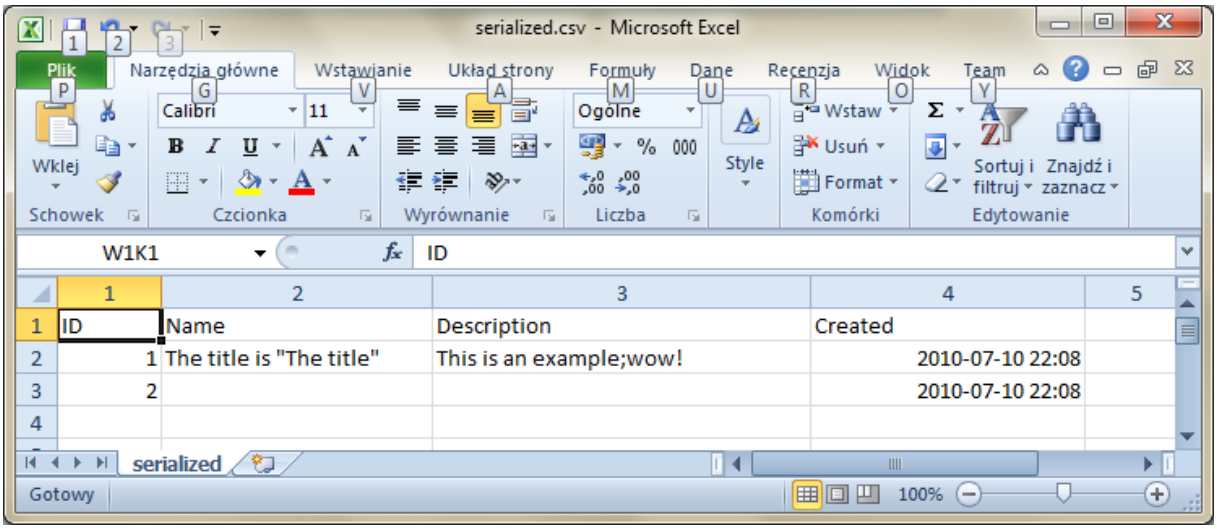
Now let's serialize our class to a CSV file named ***serialized.csv*** . The example is presented below:

```
01    using (SimpleCSVSerializer<MyObject> serializer = new
02      SimpleCSVSerializer<MyObject>()) {
03      using (SimpleCSVWriter writer = new SimpleCSVWriter(@".\serialized.csv")) {
04        writer.HasHeader = true;
05        writer.QuoteAll = true;
06        serializer.Serialize(writer, new MyObject {
07          ID = 1,
08          Title = @"The title is ""The title""",
09          Data = "This is an example;wow!",
```

```
10          Date = DateTime.Now,
11          NotForExport = "My specified data"
12        });
13        serializer.Serialize(writer, new MyObject {
14          ID = 2,
15          Date = DateTime.Now,
16           NotForExport = "No more!"
17        });
18      }
19    }
```

On line 4 we are defining, that our CSV file will contain a header. In this example the header will be generated from the class. Line 5 determines, that all cells will be Quoted in the file. Let's look on results viewed in notepad and in Microsoft Excell 2010.



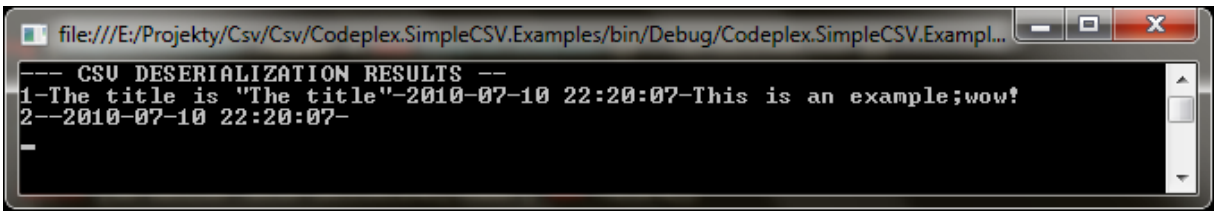


Deserializing

Deserializing is similar to serializing. Look at the example below which uses the same serializer instance like in previous example:

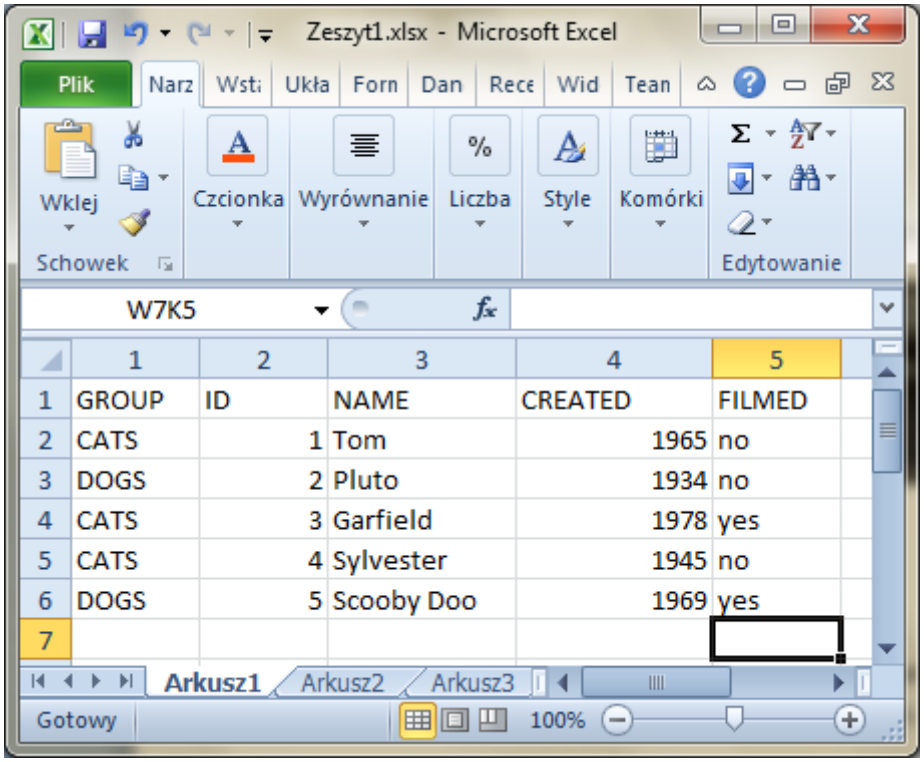
```
01  using (SimpleCSVReader reader = new SimpleCSVReader(@".\serialized.csv")) {
02    reader.HasHeader = true;
03    MyObject record;
04
05    while (serializer.DeserializeLine(reader, out record)) {
06      Console.WriteLine("{0}-{1}-{2}-{3}",
07                        record.ID,
08                        record.Title,
09                        record.Date,
10                        record.Data);
11    }
12  }
```

The result, presented on the screen below shows correctly these values:



Modifying serialization

Sometimes the data should be modified before and after serialization or deserialization. For example to map values or change the data representation. Imagine, that You have an CSV file with 5 columns like below:



This CSV file represents a list of cartoons grouped by animal type (CATS or DOGS) some cartoon ID, name, date when the cartoon was created and info if the cartoon was filmed .

It's more readable for human to view text data with some meaning like CATS or DOGS nor than numbers 1 or 2 or info yes/no rather than true/false. Let's look on object representation of one row

```
01        public class CartoonItem
02        {
03            [SimpleCSV(Label="GROUP")]
04            public int GroupID { get; set; }
05
06            [SimpleCSV]
07            public int ID { get; set; }
```

```
08
09          [SimpleCSV(Label = "NAME")]
10          public string Name { get; set; }
11
12          [SimpleCSV(Label = "CREATED", Format = "yyyy")]
13          public DateTime Created { get; set; }
14
15          [SimpleCSV(Label = "FILMED")]
16          public bool Filmed { get; set; }
17      }
```

You can see some properties labeled with SimpleCSV attribute. Look at line 12, where is presented additional attribute property named Format, where You can place the DateTime format, in what will be serialized or deserialized.[1]

Now we should be able to change integer GroupID and bool Filmed properties to string values that represent group name and yes/no values rather than true/false. To enable modifying those values implement ISimpleSerializationCallback interface like below

```
01  public class CartoonItem: ISimpleCSVSerializationCallback
02  {
…           …
17    #region ISimpleCSVSerializationCallback Members
18
19    public bool SimpleCSVDeserialization(SimpleCSVReader reader,
20                SimpleCSVSerializationState state, object handler)
21    {
22          …
23    }
24
25    public bool SimpleCSVSerialization(SimpleCSVWriter writer,
26                SimpleCSVSerializationState state, object handler)
27    {
28          …
29    }
30
31    #endregion
32  }
```

This interface enables You to modify serialization (SimpleCSVSerialization) and deserialization (SimpleCSVDeserialization) of Your class to CSV representation. To explain more this functions we will implement SimpleCSVSerialization. First let's explain the handler attribute in the function. It's represents user defined class that will be used during serialization and deserialization. This class can define some data source in our example this will be CartoonDataHanlder with two mapping methods.

```
01  public class CartoonItem: ISimpleCSVSerializationCallback
02  {
…           …
17
18    public bool SimpleCSVSerialization(SimpleCSVWriter writer,
19                SimpleCSVSerializationState state, object handler)
20    {
21      if (state == SimpleCSVSerializationState.AfterSerialization)
22      {
23        writer[1] = (handler as CartoonDataHandler).MapIDToGroup(GroupID);
24        writer[5] = Filmed ? "yes" : "no";
25      }
26
```

[1] Full list of available values is presented on http://msdn.microsoft.com/en-us/library/8kb3ddd4.aspx

```
27        return true;
28      }
29    }
```

Like You see we are mapping yes/no values (from column 5) and group ID to human readable label in column 1. Returning true in means, that the CSV line was serialized properly if You return false then serializer will return nothing (this function is useful when You will filter serialization).

Now let's examine the opposite function – SimpleCSVDeserialization. It looks similarly to SimpleCSVSerialization the difference is, that we are using writer rather than reader function argument, we are mapping from ID to Label and from true/false to yes/no.

```
01    public class CartoonItem: ISimpleCSVSerializationCallback
02    {
…                 …
30      public bool SimpleCSVSerialization(SimpleCSVWriter writer,
31                  SimpleCSVSerializationState state, object handler)
32      {
33        if (state == SimpleCSVSerializationState.AfterSerialization)
34        {
35          writer[1] = (handler as CartoonDataHandler).MapIDToGroup(GroupID);
36          writer[5] = Filmed ? "yes" : "no";
37        }
38        return true;
39      }
40    }
```

State AfterSerialization means, that the serializer has serialized objects to string representations – this is the best way to change and modify values before saving them to stream.

Using Serializer Events

If You will not implement ISimpleCSVSerializationCallback interface in class but will still be able to modify serialization and deserialization You, then the best way is to add events to serializerEvents

Event	Meaning
OnDeserialization	Event raised when CSV is deserialized
OnSerialization	Event raised when object is serialized to CSV

An short example is presented below (using lambda expressions):

```
01    serializer.OnDeserialization += (sender, eventArgs) =>
02    {
03      if (eventArgs.State == SimpleCSVSerializationState.BeforeDeserialization)
04      {
05        return "yes".Equals(eventArgs.CSVStream["FILMED"]);
06      }
07      return true;
08    };
```

Events are good example to enable CSV filtering. Above we are deserializing only records (cartoons) that has ***yes*** value in ***FILMED*** column.

Enumeration serialization

SimpleCSV allows You to serialize enumerations too. If Your class contains a custom enum type, then You can label enum values with SimpleCSVEnum like in example below:

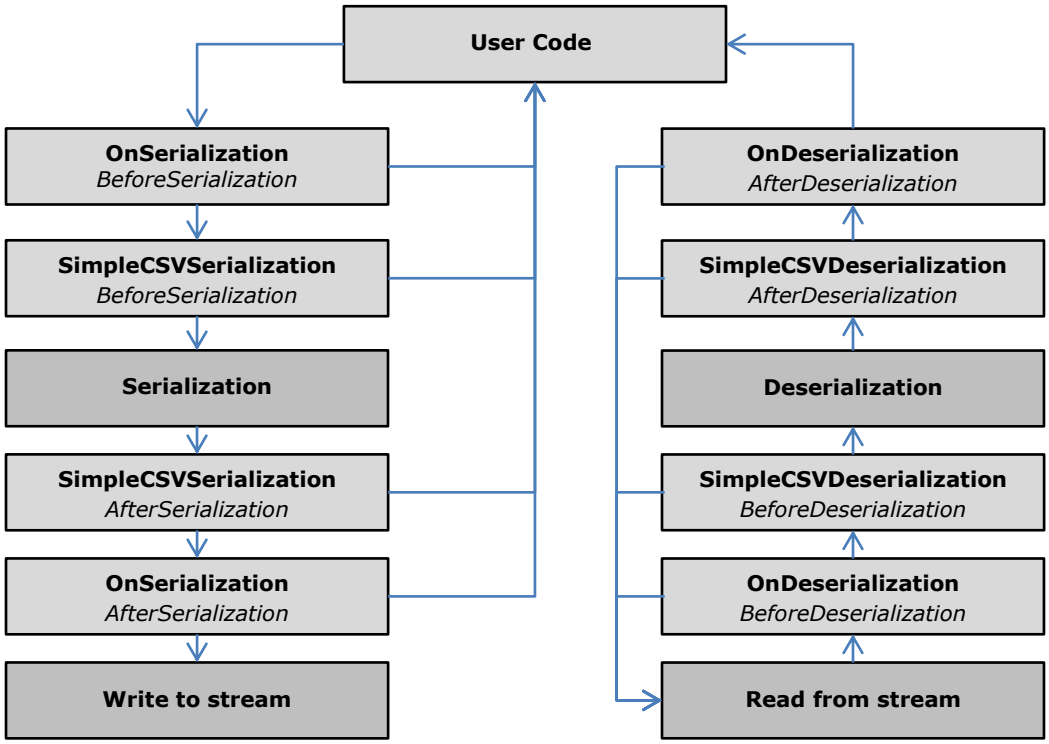
```
    public enum MyEnum
    {
        [SimpleCSVEnum(Label="GROUP_A")]
        GroupA,

        GroupB,

        [SimpleCSVEnum(Label = "GROUP_C")]
        GroupC
    }
```

Serialization Graph

This graph presents what methods or events are executed (in what order) during serialization and deserialization. During serialization, first is executed OnSerialization with BeforeSerialization status then SimpleCSVSerialization (implemented by ISimpleCSVSerializationCallback) and so on. In serialization, if You return false then the serialization is ignored and nothing is saved to stream. In deserialization if You return false then serializer will read from stream until end of stream occurs or if true is returned in deserialization callback methods or events (exceptions of course has higher status and will be returned to the nearest catch).



Other useful examples

Change the CSV column separator

To change the CSV column separator use the Splitter property of SimpleCSVReader and SimpleCSVWriter that's accepts a character. This property ***can be changed only before any read and write***.

```
01    using (SimpleCSVWriter writer = new SimpleCSVWriter(@".\simple.csv")) {
02      writer.Splitter = ',';
…
```