

# Yaml Library for .NET

Date

2006-06-03

Authors

[Christophe Lambrechts](#), [Jonathan Slenders](#)

Content

- [Introduction](#)
- [License](#)
- [Currently supported](#)
- [Get the source](#)
- [Compiling this library](#)
  - [Preprocessor definition](#)
  - [Integration in other projects](#)
- [Code examples](#)
  - [Reading YAML from a file](#)
  - [Reading YAML from a string](#)
  - [Writing YAML to a file](#)
  - [Writing YAML to a string](#)
  - [Creating a YAML tree](#)
  - [Traversing a YAML tree](#)
- [Description of our algorithms](#)
  - [Levels in our parser](#)
  - [Determining the type of node](#)
  - [Parsing of Sequences](#)
  - [Parsing of Mappings](#)
  - [Parsing of the scalars](#)
- [Documentation](#)
- [Our vision about YAML and its applications](#)
- [Credits](#)

## Introduction

Dear Yaml fans,

We are two computer science students from the [University of Hasselt](#). In our second Bachelor year we had to implement a .NET [Yaml](#) parser as a project. The requirements of the project were to implement it in C# and release it as Open Source. Further we took also the challenge to make it a one pass parser.

We will follow the project, but due of our busy student life we can't say at the moment that we will have time to do a lot of coding.

We hope you enjoy our work and it is useful.

## License

We released our software as an OpenSource project. This is one of the project requirements and also has our full support. We used the GNU General Public License license that can be found in `COPYING` file placed in the `Code` directory. More information can be found on <http://www.gnu.org/>.

## Currently supported

Yaml is a very extensive markup/serialisation language. There are a lot of parsers available over the net, and not all of them support all possible structures defined in the official documentation. We had very few time while developing this parser and so we were far from able to complete everything. We also didn't aim to complete everything because there are discussions going on about simplifying the Yaml language definition and moving to JSON compatibility. We stayed close to the core without going too much in detail. As a result of using an object oriented language, our parser is very well structured and easy to extend.

This is a list of the currently supported features.

- Sequences
- Explicit mappings
- Implicit mappings (without leading colon) (still unstable)
- Inline mappings and sequences
- Nesting of all kind of supported structures except some nestings inside inline mappings or sequences
- Strings
- Folded and block scalars (but no + and - chomps)
- Booleans
- Null values
- Integers
- Floats
- Timestamps
- Binary

This has yet to come.

- Multiple Yaml documents in a single file
- Anchors and references
- Indexers for mappings. But in a way it can be used just like a hashtable
- Some bug fixes
- ...

## Get the source

We use CVS to manage our code. You can check out with an anonymous account at this place: `ipserver:anonymous@lummba.uhasselt.be:/home/christophe/Pro2` in the "yaml" repository. At this time there are some junk files like our analyse and end report for the educational team (this is in dutch). Some files of interest are the Linux/Unix Makefile and the Visual Studio project files. Jonathan is a Linux developer/user and Christophe likes Windows. This is the reason that our library works fine on the two platforms.

## Compiling this library

### Preprocessor definition

We use preprocessor defines for easily enabling or disabling support for certain datatypes.

General

- UNSTABLE (Also enable unstable code, more features)

Node.cs

- SUPPORT\_EXPLICIT\_TYPE
- SUPPORT\_IMPLICIT\_MAPPINGS (requires UNSTABLE)

Scalar.cs

- SUPPORT\_NULL\_NODES
- SUPPORT\_INTEGER\_NODES
- SUPPORT\_FLOAT\_NODES
- SUPPORT\_BOOLEAN\_NODES
- SUPPORT\_BINARY\_NODES
- SUPPORT\_TIMESTAMP\_NODES

The precompiled DLL has everything enabled. Possibly in the future there comes a feature to enable/disable this at runtime.

This library can be compiled with the Microsoft C# .NET compiler and with Mono. We've chosen to use only the .NET 1.0 and not the new 2.0 version. This for extra compatibility with [Mono](#).

### Integration in other projects

There are two options. You can put all the code in your project or use our library as a DLL. You can find a debug and a release version in the folder `Code/bin/`.

Our project is also signed with a public key. You can find it in the same directory as the source code. We hope this will help for further development.

## Code examples

You'll have to import the YAML namespace in order to use this classes. We leave it behind in the the following code examples.

```
using Yaml;
```

If you are using Visual Studio, then you also need to refer to the folder where the Yaml DLL is placed. In the provided examples is referred to the Debug DLL found in `Code/bin/Debug/`.

Note: all this code examples and more can be found in the [Examples](#) folder.

### Reading YAML from a file

```
Node node = Node.FromFile ("testRead.yaml");
Console.WriteLine (node);
```

Note: 'Node.ToString ()' does not return YAML code. Use 'Node.Write ()' instead.

### Reading YAML from a string

```
Node node = Node.Parse ("~ item1\n~ item2\n");
Console.WriteLine (node);
```

The output looks like this:

```
[SEQUENCE][STRING]item1[/STRING][STRING]item2[/STRING][SEQUENCE]
```

### Writing YAML to a file

```
Node node = Node.Parse ("~ item1\n~ item2\n");
node.ToFile ("testWrite.yaml");
```

### Writing YAML to a string

```
Node node = Node.Parse ("~ item1\n~ item2\n");
string s = node.Write ();
Console.WriteLine( s );
```

The output looks like:

```
- "item1"
- "item2"
```

### Creating a YAML tree

```
Sequence sequence = new Sequence (
    new Node []
    {
        new Yaml.String ("item 1"),
        new Yaml.String ("item 2"),
        new Yaml.String ("item 3"),

        new Mapping (
            new MappingNode []
            {
                new MappingNode (new Yaml.String ("key 2"), new Yaml.String ("value 1")),
                new MappingNode (new Yaml.String ("key 2"), new Yaml.String ("value 2"))
            } ),

        new Yaml.String ("item 5")
    } );
Console.WriteLine (sequence);
```

The output:

Node, Sequence, Mapping, MappingNode and String are all members of the Yaml namespace. We recommend using Yaml.Node, ... Yaml.String to avoid conflicts with other namespaces.

If you are using the System namespace at the sametime, then there is a name collision with String. Use Yaml.String to solve this.

### Browse a YAML tree

```
foreach (Node s in sequence.Nodes)
{
    if (s.Type == NodeType.String)
        Console.Write ("Found a string: " + ((String) s).Content + "\n");
}

Found a string: item 1
Found a string: item 2
Found a string: item 3
Found a string: item 5
```

## Description of our algorithms

We didn't copy any algorithm from an existing parser. The main reason for doing so is because the Syck parser is written in C while we took a more object oriented approach. Other reasons include that not all the available parsers give the same parse result. Also, because it was a school project and we had to learn from this project, it had no sense to simply translate an existing parser to c#.

### Levels in our parser

Description of what we call the 'ParseStream'. See 'ParseStream.cs'.

1. Preprocessor
2. Indentation processor
3. Buffer
4. Multilevel buffer (nested buffer)
5. Comment remover
6. The-last-new-line dropper
7. Inline processor (StopAt char x)

The preprocessor's job is mainly to remember the current line number. The line number is only used to throw ParseException which contain only the line where something went wrong.

We're using here recursion. We have some kind of indentation processor which makes it possible for a nested YAML element to be parsed just like we would do at the root level, without having to care about its level of indentation or the way it appears.

Our parser is almost 100% a one pass parser. We can read from a string but also from any kind of stream without ever having to look back before the current position. There are annoying YAML structures which require us to look back. For instance, mappings doesn't require to be preceded by a question mark, so actually we don't know when we start such an implicit mapping. The parser should do some kind of trial and error to guess the type of node we are parsing. We use a circular buffer of 1024 just like described in the Yaml documentation to do some lookahead to guess and rewind if we were wrong. Implicit keys may not be longer then 1024, so the small size won't be any problem. Later on this buffer becomes also useful while guessing the kind of scalar.

While trying to parse an implicit mapping we may need to guess what type of scalar the key would be. This requires a buffer inside the buffer, this layer doesn't create a new buffer, but remembers the current position in the already existing buffer and uses a stack of start positions to restore the position while rewinding.

Another layer removes the comments.

The last newline of a block is never part of the node content, but is only necessary to start a new node. This newline will be dropped.

Inline sequences and mappings don't depend on indentation. They start with an opening "[" or "[" and stop at the matching bracket. This layer adds a method to set the chars where the parser should stop parsing.

Note that we use corresponding methods. Each call for the indent method **must** go hand in hand with an unindent method. In the same way a call for a StopAt must be followed later on with calling the DontStop method.

Most of this levels implement this methods:

Next

Move to the next character in the stream

Char

Return the current character

EOF (end of file)

True when we reached the end of the stream or substream

Each of the previously described layers derives from the previous layer and overrides the parents functions. Other parts of the parser only interact with the last layer.

## More detailed description of the indentation processor

The indentation processor has two important public methods: 'Indent' and 'UnIndent'. Calling the first method implies that we are going into a childnode, probably more indented,

```
key:
  value
```

but possibly the child is positioned directly after the parent node like in:

```
key: value
```

The indent function sets an indentation-request variable. When we continue moving through the stream and we meet a newline, the indentation processor checks how much the indentation is. If the indentation is more then the indentation of the previous line, than this line indeed belongs to the child node. Otherwise, this layer pretends that this is the end of the stream and set the EOF. Higher levels are fooled that the stream has ended until the UnIndent method has been called. When the indentation is more, we save the current level of indentation to be restored later on. Calling the UnIndent function will cancel the indentation request when we didn't meet a newline, otherwise we unpop the last indentation level from the stack.

## More detailed description of the buffer

We use a circular buffer (like a queue) with a size of 1k chars like described in the YAML documentation. Lookahead over more then 1024 characters is not possible, otherwise we could not call this a one-pass-parser. There are three important integer variables in the buffer class: rotation, position and size. Rotation points to the first cell of the buffer. Position is the offset of the current character in the buffer, so the actual position is  $(rotation + position) \% 1024$ . Size is the length of the buffer part which is currently in use.

When calling the BuildLookaheadBuffer method, the buffer will be initialized unless a build request has been done before. A DestroyBuffer call does not actually destroy the buffer, it only rotates the buffer. The current position becomes the rotation index and then it's set to zero. At that time only this layer knows that the buffer is still in use. Each Next call rotates the buffer more and more, until finally the buffer size becomes zero and it's really destroyed. This is necessary to allow destroying while not being at the end of the buffer. The Rewind call simply resets the position.

Suppose that we build a buffer but a nested node requires us to do some lookahead again. This is possible, and therefore we have a stack (in the multilevelbuffer layer) which remembers all the start positions. Rewinding restores the stack peek. Destroying un pops the stack peek.

## More detailed description of the inline processor

For instance, "{" indicates the start of an inline mapping. The parser should stop at a matching "}" or at a comma where a new item begins.

```
[aaa, bbb, ccc, {ddd: eee, "f:ff": ggg}]
```

A call of the StopAt method makes the inline processor stop at one of the given characters. We use a stack to override the 'parent' his behavior (e.g. in nested inline mappings or sequences). Note that a call of StopAt can enlarge its character stream in contrast with Indent who only reduces its. The don't stop method cancels the StopAt behavior, and restores the last setting.

We use this stop mechanism to parse mappings, inline sequences and quoted strings.

## Determining the type of node

### Explicit type indication

Nodes starting with a type indication (e.g. !str) are easy to parse, they require no lookahead. At the moment we know that a node starts with two exclamation marks, we can be sure that a type indication follows. After reading the typename, we call the parse method for that specific kind of node.

### Guessing

When no type indication is given, we have to guess what kind of node follows. A hyphen (whether or not it must be followed by a space is still a heavy discussion, we don't require it at this time) or a opening square bracket is the start of a mapping. A question mark or opening curled bracket is the start of a mapping. Other types requires lookahead via a buffer.

The next thing we try are inline mapping. We do an indentation request and set StopAt(colon), parse recursively the key and call DontStop and UnIndent. If the current character at that moment is a colon, then we're talking about an implicit mapping.

When the node wasn't an inline mapping we try all the scalars one after one. The last scalar to try is a string.

## Parsing of the scalars

Yaml is made to map data types on the build in language types. We made for each scalar an own wrapper class derived from Node. Further it is possible to represent scalars that are not jet supported, for example the time zone of a timestamp. (This is not completely correct. A time zone is supported in .NET 2.0, but for compatibility reasons we only use .NET 1.0)

We also extend the normal number range. Integers are stored in a long type, so numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 can be used.

If there is a scalar expected then a ParseStream is given to the constructor of the Scalar class. Here we require guessing like described in the previous section. We used <http://yaml.org/type/> as the reference. A important note is that there is not used the regular expression for pattern matching because of it's lack of efficiency and the loss of being an one-pass parser.

## Documentation

We used [NDoc](#) for generating code documentation. It is based on the XML comment system of .NET and gives the possibility to generate documentation in different formats.

You can explore this documentation by using the menu on the left. In this documentation you can also find the private members of the classes. If you only want to use our parser, you may not need this. We leave it here, because it is useful for development and good understanding of the parser.

For some more information about Yaml we refer to the official site <http://www.yaml.org/>.

Other useful links are:

- <http://whytheluckystiff.net/syck/> Yaml for Ruby, Python, PHP and OCaml
- <http://yaml4r.sourceforge.net/cookbook/> Yaml Cookbook
- <http://yaml.kwiki.org/?YamlInFiveMinutes> Yaml in five minutes
- <http://yaml.kwiki.org/index.cgi?YamlTheSecondFiveMinutes> Yaml in five minutes, the second five minutes


## Our vision about YAML and its applications

Yaml is easy to read and understand. It's easy to write, but in our opinion users have too much freedom in choosing how they would write the same thing. This - what looks a rich set of data types and ways to write the same - results in more confusing rather then it helps. We agree with the YAML community that there's a need to simplify the language description.

A big advantage of Yaml over XML is its support for anchors and references to this anchors. This supports not only tree-based data structures but also cyclic and acyclic data structures.

## Credits

Special thanks goes to the educational team (Tom Van Laerhoven and Jo Vermeulen). Also we will thank Gert Van Gool and Ingo Berben who gives ous some good advice on C# programming. They were doing at the same time another C# project.

 <a target="\_blank" href="http://www.webstats4u.com/stats?AD2MEgaJGhibO4bO4zXbMeL6+scw">  </a><br>