NanoXML/Lite 2.2

Marc De Scheemaecker <cyberelf@mac.com>

April 3, 2002

Contents

Chapter 1

Introduction

This chapter gives a short introduction to XML and NanoXML.

1.1 About XML

The extensible markup language, XML, is a way to mark up text in a structured document.

XML is a simplification of the complex SGML standard. SGML, the Standard Generalized Markup Language, is an international (ISO) standard for marking up text and graphics. The best known application of SGML is HTML.

Although SGML data is very easy to write, it's very difficult to write a generic SGML parser. When designing XML however, the authors removed much of the flexibility of SGML making it much easier to parse XML documents correctly.

XML data is structured as a tree of *entities*. An entity can be a string of character data or an element which can contain other entities. Elements can optionally have a set of attributes. Attributes are key/value pairs which set some properties of an element.

The following example shows some XML data:

```
<book>
  <chapter id="my chapter">
    <title>The title</title>
    Some text.
  </chapter>
</book>
```

At the root of the tree, you can find the element "book". This element contains one child element: "chapter". The chapter element has one attribute which maps the key "id" to "my chapter". The chapter element has two child entities: the element "title" and the character data "Some text.". Finally, the title element has one child, the string "The title".

1.2 About NanoXML

In April 2000, NanoXML was first released as a spin-off project of AUIT, the Abstract User Interface Toolkit.

The intent of NanoXML was to be a small parser which was easy to use. SAX and DOM are much too complex for what I needed and the mainstream parsers were either much too big or had a very restrictive license.

NanoXML 1 has all the features I needed: it is very small (about 6K), is reasonably fast for small XML documents, is very easy to use and is free (zlib/libpng license). As I never intended to use NanoXML to parse DocBook documents, there was no support for mixed data or DTD parsing.

NanoXML was released as a SourceForge project and, because of the very good response from its users, it matured to a small and stable parser. The final version, release 1.6.8 was released in May 2001.

Because of its small size, people started to use NanoXML for embedded systems (KVM, J2ME) and kindly submitted patches to make NanoXML work in such restricted environment.

1.3 NanoXML 2

In July 2001, NanoXML 2 has been released. Unlike NanoXML 1, speed and XML compliancy were considered to be very important when the new parser was designed. NanoXML 2 is also very modular: you can easily replace the different components in the parser to customize it to your needs. The modularity of NanoXML 2 also benefits extensions like e.g. SAX support which can now directly access the parser. In NanoXML 1, the SAX adapter had to iterate the data structure built by the base product.

Although many features were added to NanoXML, the second release was still very small. The full parser with builder fits in a JAR file of about 32K. This is still very tiny, especially when you compare this with the "standard" parsers of more than four times its size.

As there is still need for a tiny parser like NanoXML 1, there is a special branch of NanoXML 2: NanoXML/Lite. This parser is source compatible with NanoXML 1 but features a new parsing algorithm which makes it more than twice as fast as the older version. It is however more restrictive on the XML data it parses: the older version allowed some not-wellformed data to be parsed.

There are three branches of NanoXML 2:

- *NanoXML/Lite* is the successor of NanoXML 1. It features an almost compatible parser which is extremely small.
- *NanoXML/Java* is the standard parser.
- *NanoXML/SAX* is the SAX adapter for NanoXML/Java.

The latest version of NanoXML is NanoXML 2.2.1, which has been released in February 2002.

Chapter 2

Retrieving Data From An XML Datasource

This chapter shows how to retrieve XML data from a standard data source. Such source can be a file, an HTTP object or a text string.

2.1 A Very Simple Example

This section describes a very simple XML application. It parses XML data from a stream and dumps it to the standard output. While its use is very limited, it shows how to set up a parser and parse an XML document.

```
import nanoxml.*;                                                    ①
import java.io.*;
public class DumpXML
{
  public static void main(String[] args)
    throws Exception
  {
    XMLElement xml = new XMLElement();                               ②
    FileReader reader = new FileReader("test.xml");
    xml.parseFromReader(reader);                                     ③
    System.out.println(xml);                                         ④
  }
}
```

① The NanoXML classes are located in the package *nanoxml*.

② This command creates an empty XML element.

③ The method `parseFromReader` parses the data in the file *test.xml* and fills the empty element.

④ The XML element is dumped to the standard output.

2.2 Analyzing The Data

You can easily traverse the logical tree generated by the parser. By calling one of the `parse*` methods, you fill an empty XML element with the parsed contents. Every such object can have a name, attributes, #PCDATA content and child objects.

The following XML data:

```
<FOO attr1="fred" attr2="barney">
  <BAR a1="flintstone" a2="rubble">
    Some data.
  </BAR>
  <QUUX/>
</FOO>
```

is parsed to the following objects:

Element FOO:

Attributes = { "attr1"="fred", "attr2"="barney" }

Children = { BAR, QUUX }

PCData = null

Element BAR:

Attributes = { "a1"="flintstone", "a2"="rubble" }

Children = {}

PCData = "Some data."

Element QUUX:

Attributes = {}

Children = {}

PCData = null

You can retrieve the name of an element using the method `getName`, thus:

```
FOO.getName() → "FOO"
```

You can enumerate the attribute names using the method `enumerateAttributeNames`:

```
Enumeration enum = FOO.enumerateAttributeNames();
while (enum.hasMoreElements()) {
  System.out.print(enum.nextElement());
  System.out.print(' ');
}
→ attr1 attr2
```

You can retrieve the value of an attribute using `getAttribute`:

```
FOO.getAttribute("attr1") → "fred"
```

The child elements can be enumerated using the method `enumerateChildren`:

```
Enumeration enum = FOO.enumerateChildren();
while (enum.hasMoreElements()) {
```

```
  XMLElement child = (XMLElement) enum.nextElement();
  System.out.print(child.getName() + ' ');
}
→ BAR QUUX
```

If the element contains parsed character data (#PCDATA) as its only child. You can retrieve that data using `getContent`:

```
BAR.getContent() → "Some data."
```

Note that in NanoXML/Lite, a child cannot have children and #PCDATA content at the same time.

2.3 Generating XML

You can very easily create a tree of XML elements or modify an existing one. To create a new tree, just create an `XMLElement` object:

```
XMLElement elt = new XMLElement("ElementName");
```

You can add an attribute to the element by calling `setAttribute`:

```
elt.setAttribute("key", "value");
```

You can add a child element to an element by calling `addChild`:

```
XMLElement child = new XMLElement("Child");
elt.addChild(child);
```

If an element has no children, you can add #PCDATA content to it using `setContent`:

```
child.setContent("Some content");
```

Note that in NanoXML/Lite, a child cannot have children and #PCDATA content at the same time.

When you have created or edited the XML element tree, you can write it out to an output stream or writer using the method `toString`:

```
java.io.PrintWriter output = ...;
XMLElement xmltree = ...;
output.println(xmltree);
```