NanoXML/Java 2.0

Marc De Scheemaecker

Contents

CHAPTER 1

Introduction

This chapter gives a short introduction to XML and NanoXML.

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".

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.

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.

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. The method described in this chapter is the simplest way to retrieve XML data. More advanced ways are described in the next chapters.

A Very Simple Example

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

```
import net.n3.nanoxml.*;                                      //1
import java.io.*;

public class DumpXML
{
    public static void main(String[] args)
        throws Exception
    {
        IXMLParser parser                                     //2
             = XMLParserFactory
```

```
                  .createDefaultXMLParser();
        IXMLReader reader                                   //3
            = StdXMLReader.fileReader("test.xml");
        parser.setReader(reader);
        XMLElement xml                                      //4
            = (XMLElement) parser.parse();
        XMLWriter writer                                    //5
            = new XMLWriter(System.out);
        writer.write(xml);
    }
}
```

(1) The NanoXML classes are located in the package net.n3.nanoxml.

(2) This command creates an XML parser. The actual class of the parser is dependent on the value of the system property `net.n3.nanoxml.XMLParser`, which is by default `net.n3.nanoxml.StdXMLParser`.

(3) The command creates a "standard" reader which reads its data from the file called *test.xml*.

Usually you can use `StdXMLReader` to feed the XML data to the parser. The default reader is able to set up HTTP connections when retrieving DTDs or entities from different machines. If necessary, you can supply your own reader to e.g. provide support for `PUBLIC` identifiers.

(4) The XML parser now parses the data read from *test.xml* and creates a tree of parsed XML elements.

The structure of those elements will be described in the next section.

(5) An *XMLWriter* can be used to dump a "pretty-printed" view of the parsed XML data on an output stream. In this case, we dump the read data to the standard output (`System.out`).

Analyzing The Data

You can easily traverse the logical tree generated by the parser. If you need to create your own object tree, you can create your custom builder, which is described in chapter 3.

The default XML builder, `StdXMLBuilder` generates a tree of `XMLElement` objects. Every such object has a name and can have 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 in 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 keys 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:

```
FOOgetAttribute ("attr1") → "fred"
```

The child elements can be enumerated using the method enumerateChildren:

```
Enumeration enum = FOO.enumerateChildren();
while (enum.hasMoreElements()) {
    System.out.print(enum.nextElement() + ' ');
}

    → 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."
```

If an element contains both `#PCDATA` and XML elements as its children, the character data segments will be put in untitled XML elements (whose name is null).

`XMLElement` contains many convenience methods for retrieving data and traversing the XML tree. You can find them on page 66.

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("ChildElement");
elt.addChild(child);
```

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

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

If the element does have children, you can add #PCDATA content to it by adding an untitled element:

```
XMLElement pcdata = new XMLElement();
pcdata.setContent("Blah blah");
elt.addChild(pcdata);
```

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

```
java.io.Writer output = ...;
XMLElement xmltree = ...;
XMLWriter xmlwriter = new XMLWriter(output);
writer.write(xmltree);
```

CHAPTER 3

Retrieving Data From An XML Stream

If you're retrieving data from a stream, but you don't want to wait to process the data till it's completely read, you can use *streaming*.

The XML Builder

The XML data tree is created using an *XML builder*. By default, the builder creates a tree of `XMLElement`.

While the parser parses the data, it notifies the builder of any elements it encounters. Using this information, the builder generate the object tree. When the parser is done processing the data, it retrieves the object tree from the builder using *getResult*.

The following example shows a simple builder that prints the notifications on the standard output.

```
import java.io.*;
import net.n3.nanoxml.*;

public class MyBuilder
    implements IXMLBuilder
{
```

```
public void startBuilding(String systemID,      //1
                          int    lineNr)
{
    System.out.println("Document started");
}

public void                                     //2
        newProcessingInstruction(String target,
                                 Reader reader)
    throws IOException
{
    System.out.println("New PI with target "
                       + target);
}

public void startElement(String name,           //3
                         String nsPrefix,
                         String nsSystemID,
                         String systemID,
                         int    lineNr)
{
    System.out.println("Element started: " + name);
}

public void endElement(String name,             //4
                       String nsPrefix,
                       String nsSystemID)
{
    System.out.println("Element ended: " + name);
}

public void addAttribute(String key,            //5
                         String nsPrefix,
                         String nsSystemID,
                         String value,
                         String type)
{
    System.out.println("  " + key + ": "
                       + type + " = " + value);
}

public void elementAttributesProcessed(         //6
        String name,
        String nsPrefix,
        String nsSystemID)
    {
        // nothing to do
    }

    public void addPCData(Reader reader,                //7
                          String systemID,
                          int    lineNr)
        throws IOException
    {
        System.out.println("#PCDATA");
    }

    public Object getResult()                           //8
    {
        return null;
    }
}
```

(1) The XML parser started parsing the document. The *lineNr* parameter contains the line number where the document starts.

(2) The XML parser encountered a processing instruction (PI) which is not handled by the parser itself. The *target* contains the target of the PI. The contents of the PI can be read from *reader*.

(3) A new element has been started at line *lineNr*. The name of the element is *name*.

(4) The current element has ended. For convenience, the name of that element is put in the parameter *name*.

(5) An attribute is added to the current element.

(6) This method is called when all the attributes of the current element have been processed.

(7) A #PCDATA section has been encountered. The contents of the section can be read from *reader*.

(8) This method is called when the parsing has finished. If the builder has a result, it has to return it to the parser in this method.

Registering an XML Builder

You can register the builder to the parser using the method *setBuilder*.

The following example shows how to create a parser which uses the builder we created in the previous section:

```
import net.n3.nanoxml.*;
import java.io.*;

public class DumpXML {
    public static void main(String args[])
            throws Exception {
        IXMLParser parser
             = XMLParserFactory
                 .createDefaultXMLParser();
        IXMLReader reader
            = StdXMLReader.fileReader("test.xml");
        parser.setReader(reader);
        parser.setBuilder(new MyBuilder());
        parser.parse();
    }
}
```

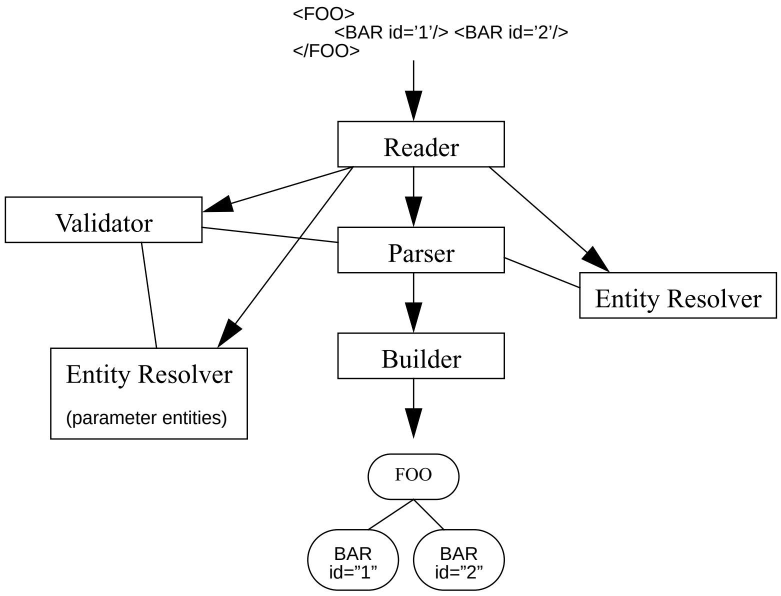
CHAPTER 4

Advanced Topics

This chapter explains how you can customize the NanoXML parser setup. Unlike NanoXML 1, NanoXML/Java 2 is designed as a *framework:* it is composed of many different components which you can plug together. It's possible to change the reader, the builder, the validator and even the parser.

NanoXML/Java comes with one set of components. Except for NanoXML/Lite, every branch offers its own set of components customized for a certain purpose. NanoXML/SAX offers components for using NanoXML as a parser for the SAX framework. NanoXML 2.1 will add two other branches: NanoXML/Adapters which will allow to plug in an existing SAX parser (like e.g. Xerces) into the framework and NanoXML/DOM which will build a DOM structure.

The following figure gives a short representation of the major components.



The *reader* retrieves data from a Java input stream and provides character data to the other components.

The *parser* converts the character data it retrieves from the reader to XML events which it sends to the builder.

The *validator* parses a DTD and validates the XML data. The current validator does only the minimum necessary for a non-validating parser. It is also not namespace aware. Version 2.1 of NanoXML/Java will implement these features.

The *entity resolvers* converts entity references (&...;) and parameter entity references (%...;) to character data. The resolver uses the reader to access external entities.

The *builder* interpretes XML events coming from the parser and builds a tree of XML elements. The standard builder creates a tree of `XMLElement`. You can provide your own builder to create a custom tree or if you are interested in the XML events themselves, e.g. to use XML streaming.

The NanoXML Reader

The reader retrieves data from some source and feeds it to the other components.

The reader is basically a stack of push-back readers. Every time a new data stream becomes active, the current reader is pushed on a stack. When the current reader has no more data left, the parent reader is popped from the stack.

If you want to implement public IDs using e.g. a catalog file similar to SGML, you could implement a reader by overriding the method *openStream* of `StdXMLReader`:

```
public class MyReader
    extends StdXMLReader
{
    private Properties publicIDs;

    public MyReader(Properties publicIDs)
    {
        this.publicIDs = publicIDs;
    }

    public Reader openStream(String publicID,
                             String systemID)
        throws MalformedURLException,
               FileNotFoundException,
               IOException
    {
        if (publicID != null) {
            systemID = publicIDs.getProperty(publicID,
                                             systemID);
        }
        return super.openStream(publicID, systemID);
    }
}
```

In this example, you have to provide a properties object which maps public IDs to system IDs.

The NanoXML Parser

The parser analyzes the character stream it retrieves from the reader and sends XML events to the builder. It uses a validator to validate the data and an entity resolver to resolve general entities. You rarely need to create a custom parser. If you need to, you have to implement `IXMLParser`.

The NanoXML Validator

The validator parses the DTD and checks the XML data. NanoXML 2.0 uses a NonValidator implementation that only performs the minimum necessary for a non-validating parser. NanoXML 2.1 will include a full-blown validator.

As a DTD is very vague, you can implement your own validator to perform a more fine-grained check of the XML data. The easiest way to create your own validator is to create a subclass of `ValidatorPlugin`.

The following example shows how to implement a validator. It checks that every attribute named “id” starts with three capital letters.

```
public class MyValidator
    extends ValidatorPlugin
{
    public void attributeAdded(String key,
                               String nsPrefix,
                               String nsSystemID,
                               String value,
                               String systemID,
                               int    lineNr)
    {
        boolean valid = true;
        if (key.equals("id")) {
            if (value.length() < 3) {
                valid = false;
            } else {
                for (int i = 0; i < 3; i++) {
                    char ch = value.charAt(i);
                    if ((ch < ’A’) || (ch > ’Z’)) {
                        valid = false;
                    }
                }
            }
        }
        if (valid) {
            super.attributeAdded(key, nsPrefix,
                                 nsSystemID, value,
                                 systemID, lineNr);
        } else {
            this.attributeWithInvalidValue(systemID,
                    lineNr, null, key, value);
    }
}
```

To register the validator to a parser, use the following code:

```
IXMLParser parser …
…
IXMLValidator val1 = parser.getValidator();
MyValidator val2 = new MyValidator();
val2.setDelegate(val1);
parser.setValidator(val2);
```

The NanoXML Entity Resolvers

The entity resolver converts entity references to XML data. If you want e.g. to retrieve entity values from a database, you have to create your own resolver.

Entity resolvers have to implement `IXMLEntityResolver`. Usually, you only have to make a subclass of `XMLEntityResolver` and implement the method *getEntity* or *openExternalEntity*.

Entities can be used in the XML data and in the DTD. As these entities are independent of each other, there are two entity resolvers.

Standard Entities

The resolver for standard entities has to be registered to the parser by calling *setResolver*. The following example registers a resolver that forces the entity "`&foo;`" to be resolved to "`bar`:"

```
import net.n3.nanoxml.*;
import java.io.*;

class MyResolver
    extends XMLEntityResolver
{
    public Reader getEntity(IXMLReader xmlReader,
                            String     name)
        throws XMLParseException
    {
        if (name.equals("foo")) {
            return new StringReader("bar");
        } else {
            return super.getEntity(xmlReader, name);
        }
    }
}

public class Demo
{
    public static void main(String[] args)
        throws Exception
    {
        IXMLParser parser
            = XMLParserFactory
                 .createDefaultXMLParser();
        parser.setResolver(new MyResolver());
        IXMLReader reader
            = StdXMLReader.fileReader("test.xml");
        parser.setReader(reader);
        XMLElement xml = (XMLElement) parser.parse();
        XMLWriter writer = new XMLWriter(System.out);
        writer.write(xml);
    }
}
```

Parameter Entities

The resolver for parameter entities has to be registered to the validator by calling *setParameterEntityResolver*. The following example show a custom version of the `Demo` class that registers `MyResolver` as a parameter entity resolver.

```
public class Demo
{
        public static void main(String[] args)
            throws Exception
        {
            IXMLParser parser
                = XMLParserFactory
                     .createDefaultXMLParser();
            IXMLValidator validator
                = parser.getValidator();
            validator.setParameterEntityResolver(
                    new MyResolver());
            IXMLReader reader
                = StdXMLReader.fileReader("test.xml");
            parser.setReader(reader);
            XMLElement xml = (XMLElement) parser.parse();
            XMLWriter writer = new XMLWriter(System.out);
            writer.write(xml);
        }
    }
```

The NanoXML Builder

The builder interpretes XML events coming from the parser and builds a tree of Java objects. When the parsing is done, the builder hands over its result to the parser.

As explained in chapter 3, the builder can also be used to read XML data while it's being streamed. This feature is useful if you don't want to wait until all the data has been read before processing the information.

As an example, we have the following XML structure (*document.dtd*):

```
<!ELEMENT Chapter (Paragraph*)>
<!ATTLIST Chapter
        title CDATA #REQUIRED
        id    CDATA #REQUIRED>
<!ELEMENT Paragraph (#PCDATA)>
<!ATTLIST Paragraph
        align (left|center|right) "left">
```

The elements are put in the Java classes `Chapter` and `Paragraph` which, for convenience, extend the following base class:

```
public class DocumentElement
{
    protected Properties attrs;
    protected Vector children;

    public DocumentElement()
    {
        this.attrs = new Properties();
        this.children = new Vector();
    }

    public void setAttribute(String attrName,
                             String value)
    {
        this.attrs.put(attrName, value);
    }

    public void addChild(DocumentElement elt)
    {
        this.children.addElement(elt);
    }
}
```

This base class simply makes it easy for our builder to set attributes and to add children to an element.

The `Chapter` and `Paragraph` classes extend this base class to give more practical access to their attributes and children:

```
public class Chapter
    extends DocumentElement
{
    public String getTitle()
    {
        return this.attrs.getProperty("title");
    }

    public String getID()
    {
        return this.attrs.getProperty("id");
    }

    public Enumeration getParagraphs()
    {
            return this.children.elements();
        }
    }

    public class Paragraph
        extends DocumentElement
    {
        public static final int LEFT = 0;
        public static final int CENTER = 1;
        public static final int RIGHT = 2;

        private static Hashtable alignments;

        static {
            alignments = new Hashtable();
            alignments.put("left", new Integer(LEFT));
            alignments.put("center", new Integer(CENTER));
            alignments.put("right", new Integer(RIGHT));
        }

        public String getContent()
        {
            return this.attrs.getProperty("#PCDATA");
        }

        public int getAlignment()
        {
            String str = this.attrs.getProperty("align");
            Integer align = alignments.get(str);
            return align.intValue();
        }
    }
```

The builder creates the data structure based on the XML events it receives from the parser. Because both `Chapter` and `Paragraph` extend `DocumentElement`, the builder is fairly simple.

```
    import net.n3.nanoxml.*;
    import java.util.*;
    import java.io.*;

    public class DocumentBuilder
        implements IXMLBuilder
    {
```

```
private static Hashtable classes;
private Stack elements;
private DocumentElement topElement;

static {
    classes = new Hashtable();
    classes.put("Chapter", Chapter.class);
    classes.put("Paragraph", Paragraph.class);
}

public void startBuilding(String systemID,
                          int    lineNr)
{
    this.elements = new Stack();
    this.topElement = null;
}

public void newProcessingInstruction(String target,
                                     Reader reader)
{
    // nothing to do
}

public void startElement(String name,
                         String nsPrefix,
                         String nsSystemID,
                         String systemID,
                         int    lineNr)
{
    DocumentElement elt = null;
    try {
        Class cls = (Class) classes.get(name);
        elt = (DocumentElement) cls.newInstance();
    } catch (Exception e) {
        // ignore the exception
    }
    this.elements.push(elt);
    if (this.topElement == null) {
        this.topElement = elt;
    }
}

public void endElement(String name,
                       String nsPrefix,
                       String nsSystemID)
```

```
{
    DocumentElement child
        = (DocumentElement) this.elements.pop();
    if (! this.elements.isEmpty()) {
        DocumentElement parent
           = (DocumentElement)this.elements.peek();
        parent.addChild(child);
    }
}

public void addAttribute(String key,
                         String nsPrefix,
                         String nsSystemID,
                         String value,
                         String type)
{
    DocumentElement child
        = (DocumentElement) this.elements.peek();
    child.setAttribute(key, value);
}

public void elementAttributesProcessed(
        String name,
        String nsPrefix,
        String nsSystemID)
{
    // nothing to do
}

public void addPCData(Reader reader,
                      String systemID,
                      int    lineNr)
    throws IOException
{
    StringBuffer str = new StringBuffer(1024);
    char[] buf = new char[bufSize];
    for (;;) {
        int size = reader.read(buf);
        if (size < 0) {
            break;
        }
        str.append(buf, 0, size);
    }
    this.addAttribute("#PCDATA", null, null,
                      str.toString(), "CDATA");
        }

        public Object getResult()
        {
            return topElement;
        }
    }
```

Note that, for simplicity, error and exception handling is not present in this example. The builder holds a stack of the current elements it builds. Character data is read from a reader. The method *addPCData* reads this data in blocks of 1K.

Finally, this application sets up the NanoXML parser and converts an XML document to HTML which it dumps on the standard output:

```
    import java.util.*;
    import net.n3.nanoxml.*;

    public class XML2HTML
    {
        public static void main(String[] params)
            throws Exception
        {
            IXMLBuilder builder
                = new DocumentBuilder();
            IXMLParser parser
                 = XMLParserFactory
                      .createDefaultXMLParser();
            parser.setBuilder(builder);
            IXMLReader reader
                = StdXMLReader.fileReader(param[0]);
            parser.setReader(reader);
            Chapter chapter = (Chapter) parser.parse();
            System.out.println("<!DOCTYPE … >");
            System.out.print("<HTML><HEAD><TITLE>");
            System.out.print(chapter.getTitle());
            System.out.println("</TITLE></HEAD><BODY>");
            System.out.print("<H1>");
            System.out.print(chapter.getTitle());
            System.out.println("</H1>");
            Enumeration enum = chapter.getParagraphs();
            while (enum.hasMoreElements()) {
                Paragraph para
                    = (Paragraph) enum.nextElement();
                System.out.print("<P>");
                System.out.print(para.getContent());
                System.out.println("</P>");
            }
            System.out.println("</BODY></HTML>");
        }
    }
```

If we run the example on the following XML file:

```
<!DOCTYPE Chapter SYSTEM "document.dtd">

<Chapter id="ch01" title="The Title">
    <Paragraph>First paragraph...</Paragraph>
    <Paragraph>Second paragraph...</Paragraph>
</Chapter>
```

The output will be:

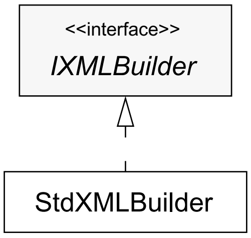
```
<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01//EN'
  'http://www.w3.org/TR/html4/strict.dtd'>
<HTML><HEAD><TITLE>The Title</TITLE></HEAD><BODY>
<H1>The Title</H1>
<P>First paragraph...</P>
<P>Second paragraph...</P>
</BODY>
```

CHAPTER 5

Class Reference

This chapter gives an overview of the Java classes.

net.n3.nanoxml.IXMLBuilder (interface)



NanoXML uses `IXMLBuilder` to construct the XML data structure it retrieved from its data source. You can supply your own builder or you can use the default builder of NanoXML. You can find more information about custom builders on page 21.

If a method of the builder throws an exception, the parsing is aborted and the *parse* method throws an `XMLException` which encapsulates the original exception.

addAttribute

```
void addAttribute(java.lang.String key,
                  java.lang.String nsPrefix,
                  java.lang.String nsSystemID,
                  java.lang.String value,
                  java.lang.String type)
    throws Exception;
```

This method is called when a new attribute of an XML element is encountered.

key	The key (name) of the attribute.
nsPrefix	The prefix used to identify the namespace. If no namespace has been specified, this parameter is `null`.
nsSystemID	The system ID associated with the namespace. If no namespace has been specified, or no system ID is associated with nsPrefix, this parameter is `null`.
value	The value of the attribute.
type	The type of the attribute. If no type is known, "CDATA" is returned.

addPCData

```
void addPCData(java.io.Reader   reader,
               java.lang.String systemID,
               int              lineNr)
    throws Exception;
```

This method is called when a PCDATA element is encountered. A Java reader is supplied from which you can read the data. The reader will only read the data of the element. You don’t need to check for boundaries. If you don’t read the full element, the rest of the data is skipped. You also don’t have to care about entities: they are resolved by the parser.

reader	The method can retrieve the data from this reader. You may close the reader before reading all its data and you cannot read too much data.
systemID	The system ID of the XML data source
lineNr	The line in the source where the element starts.

elementAttributesProcessed

```
void elementAttributesProcessed(java.lang.String name,
                                java.lang.String nsPrefix,
                                java.lang.String nsSystemID)
    throws Exception;
```

This method is called when all the attributes of an XML element have been processed.

name	The name of the element.
nsPrefix	The prefix used to identify the namespace. If no namespace has been specified, this parameter is `null`.
nsSystemID	The system ID associated with the namespace. If no namespace has been specified, or no system ID is associated with nsPrefix, this parameter is `null`.

endElement

```
void endElement(java.lang.String name,
                java.lang.String nsPrefix,
                java.lang.String nsSystemID)
    throws Exception;
```

This method is called when the end of an XML element is encountered.

name	The name of the element.
nsPrefix	The prefix used to identify the namespace. If no namespace has been specified, this parameter is `null`.
nsSystemID	The system ID associated with the namespace. If no namespace has been specified, or no system ID is associated with nsPrefix, this parameter is `null`.

getResult

```
java.lang.Object getResult()
    throws Exception;
```

Returns the result of the building process. This method is called just before the *parse* method of `IXMLParser` returns. The method has to return the result of the building process.

newProcessingInstruction

```
void newProcessingInstruction(java.lang.String target,
                              java.io.Reader   reader)
    throws Exception;
```

This method is called when a processing instruction is encountered. A PI with a reserved target ("xml" with any case) is never reported.

target	The processing instruction target.
reader	The method can retrieve the parameter of the PI from this reader. You may close the reader before reading all its data and you cannot read too much data.

startBuilding

```
void startBuilding(java.lang.String systemID,
                   int              lineNr)
    throws Exception;
```

This method is called before the parser starts processing its input.

systemID	The system ID of the XML data source
lineNr	The line on which the parsing starts.

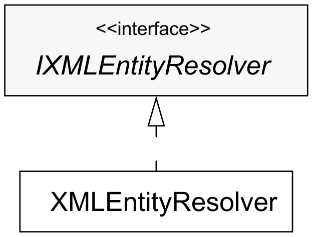
startElement

```
     void startElement(java.lang.String name,
                       java.lang.String nsPrefix,
                       java.lang.String nsSystemID,
                       java.lang.String systemID,
                       int              lineNr)
         throws Exception;
```

This method is called when a new XML element is encountered.

name	The name of the element.
nsPrefix	The prefix used to identify the namespace. If no namespace has been specified, this parameter is `null`.
nsSystemID	The system ID associated with the namespace. If no namespace has been specified, or no system ID is associated with nsPrefix, this parameter is `null`.
systemID	The system ID of the XML data source.
lineNr	The line in the source where the element starts.

net.n3.nanoxml.IXMLEntityResolver (interface)



An `IXMLEntityResolver` resolves entities. More information about custom entity resolvers can be found on page 19.

addExternalEntity

```
void addExternalEntity(java.lang.String name,
                       java.lang.String publicID,
                       java.lang.String systemID);
```

Adds an external entity.

name	The name of the entity.
publicID	The public ID of the entity.
systemID	The system ID of the entity.

addInternalEntity

```
void addInternalEntity(java.lang.String name,
                       java.lang.String value);
```

Adds an internal entity.

name	The name of the entity.
value	The value of the entity.

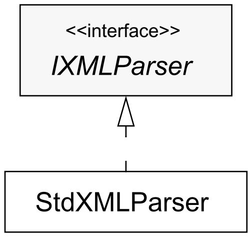
getEntity

```
java.io.Reader getEntity(IXMLReader       xmlReader,
                         java.lang.String name);
```

Returns a Java reader containing the value of an entity. If the entity could not be resolved, `null` is returned. The method may throw an `XMLParseException` if necessary.

xmlReader	The current NanoXML reader.
name	The name of the entity.

net.n3.nanoxml.IXMLParser (interface)



`IXMLParser` is the core parser of NanoXML.

getBuilder

```
IXMLBuilder getBuilder();
```

Returns the builder which creates the local structure of the XML data.

getReader

```
IXMLReader getReader();
```

Returns the reader form which the parser retrieves its data.

getResolver

```
IXMLEntityResolver getResolver();
```

Returns the entity resolver.

getValidator

```
IXMLValidator getValidator();
```

Returns the validator that validates the XML data.

parse

```
java.lang.Object parse()
    throws XMLException;
```

Parses the data and lets the builder create the logical structure. The method returns the result of *getData* of the builder described on page 32. If an error occurred while reading or parsing the data, the method may throw an `XMLException`.

setBuilder

```
void setBuilder(IXMLBuilder builder);
```

Sets the builder which creates the local structure of the XML data.

builder	The builder

setReader

```
void setReader(IXMLReader reader);
```

Sets the reader from which the parser retrieves its data.

reader	The reader

setResolver

```
void setResolver(IXMLEntityResolver resolver);
```

Sets the entity resolver.

resolver	The resolver

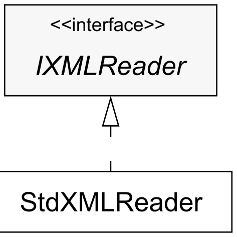
setValidator

```
    void setValidator(IXMLValidator validator);
```

Sets the validator that validates the XML data.

validator	The validator

net.n3.nanoxml.IXMLReader (interface)



`IXMLReader` reads the data to be parsed. More information about custom readers can be found on page 17.

atEOF

```
    boolean atEOF()
        throws java.io.IOException;
```

Returns `true` if there are no more characters left to be read.

atEOFOfCurrentStream

```
    boolean atEOFOfCurrentStream()
        throws java.io.IOException;
```

Returns `true` if the current stream has no more characters left to be read.

getLineNr

```
int getLineNr();
```

Returns the line number of the data in the current stream.

getPublicID

```
java.lang.String getPublicID();
```

Returns the public ID of the current stream.

getSystemID

```
java.lang.String getSystemID();
```

Returns the system ID of the current stream.

openStream

```
java.io.Reader openStream(java.lang.String publicID,
                          java.lang.String systemID)
        throws java.net.MalformedURLException,
               java.io.FileNotFoundException,
               java.io.IOException;
```

Opens a stream from a public and system ID.

publicID	The public ID of the entity (may be null)
systemID	The system ID of the entity.

read

```
char read()
    throws java.io.IOException;
```

Reads a character.

setPublicID

```
void setPublicID(java.lang.String publicID);
```

Sets the public ID of the current stream.

publicID	The public ID.

setSystemID

```
void setSystemID(java.lang.String systemID)
        throws java.net.MalformedURLException;
```

Sets the system ID of the current stream.

systemID	The system ID.

startNewStream

```
void startNewStream(java.io.Reader reader);
```

Starts a new stream from a Java reader. The new stream is used temporary to read data from. If that stream is exhausted, control returns to the "parent" stream.

reader	The reader to read the new data from.

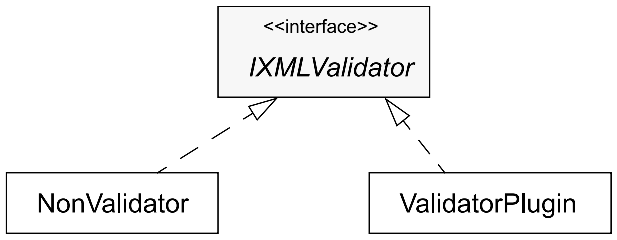
unread

```
void unread(char ch)
        throws java.io.IOException
```

Pushes the last character read back to the stream.

ch	The character to push back

net.n3.nanoxml.IXMLValidator (interface)



`IXMLValidator` processes the DTD and handles entity references. More information about custom validators can be found on page 18.

attributeAdded

```
void attributeAdded(java.lang.String key,
                    java.lang.String nsPrefix,
                    java.lang.String nsSystemID,
                    java.lang.String value,
                    java.lang.String systemID,
                    int              lineNr)
    throws java.lang.Exception;
```

Indicates that an attribute has been added.

key	The name of the attribute.
nsPrefix	The prefix used to identify the namespace
nsSystemID	The system ID associated with the namespace
value	The value of the attribute
systemID	The system ID of the XML data of the element
lineNr	The line number in the XML data of the element

elementAttributesProcessed

```
void elementAttributesProcessed(
        java.lang.String     name,
        java.lang.String     nsPrefix,
        java.lang.String     nsSystemID,
        java.util.Properties extraAttributes,
        java.lang.String     systemID,
        int                  lineNr)
    throws java.lang.Exception;
```

This method is called when the attributes of an XML element have been processed.

If there are attributes with a default value which have not been specified yet, they have to be put in *extraAttributes*.

name	The name of the element.
nsPrefix	The prefix used to identify the namespace
nsSystemID	The system ID associated with the namespace
extraAttributes	Where to put extra attributes
systemID	The system ID of the XML data of the element
lineNr	The line number in the XML data of the element

elementStarted

```
void elementStarted(java.lang.String name,
                    java.lang.String nsPrefix,
                    java.lang.String nsSystemID,
                    java.lang.String systemID,
                    int              lineNr)
    throws java.lang.Exception;
```

Indicates that an element has been started.

name	The name of the element.
nsPrefix	The prefix used to identify the namespace
nsSystemID	The system ID associated with the namespace
systemID	The system ID of the XML data of the element
lineNr	The line number in the XML data of the element

getParameterEntityResolver

```
IXMLEntityResolver getParameterEntityResolver();
```

Returns the parameter entity resolver.

parseDTD

```
void parseDTD(java.lang.String   publicID,
              IXMLReader         reader,
              IXMLEntityResolver resolver,
              boolean            external)
    throws java.lang.Exception;
```

Parses the DTD. The validator object is responsible for reading the full DTD.

publicID	The public ID, which may be null.
reader	The reader to read the DTD from.
resolver	The entity resolver.
external	`true` if the DTD is external

PCDataAdded

```
void PCDataAdded(java.lang.String systemID,
                 int              lineNr)
    throws java.lang.Exception;
```

Indicates that a new #PCDATA element has been encountered.

systemID	The system ID of the XML data of the element
lineNr	The line number in the XML data of the element

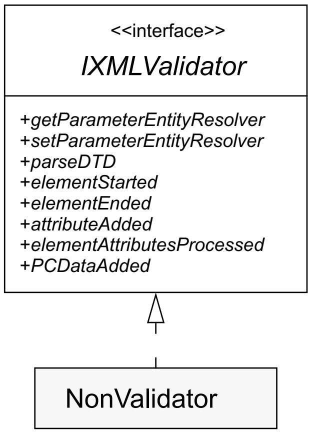
setParameterEntityResolver

```
void setParameterEntityResolver(
        IXMLEntityResolver resolver);
```

Sets the parameter entity resolver.

resolver	The resolver

net.n3.nanoxml.NonValidator



`NonValidator` is a concrete implementation of `IXMLValidator` which processes the DTD and handles entity definitions. It does not do any validation itself.

attributeAdded

```
void attributeAdded(java.lang.String key,
                    java.lang.String nsPrefix,
                    java.lang.String nsSystemID,
                    java.lang.String value,
                    java.lang.String systemID,
                    int              lineNr)
    throws java.lang.Exception;
```

Indicates that an attribute has been added.

key	The name of the attribute.
nsPrefix	The prefix used to identify the namespace
nsSystemID	The system ID associated with the namespace
value	The value of the attribute
systemID	The system ID of the XML data of the element
lineNr	The line number in the XML data of the element

elementAttributesProcessed

```
void elementAttributesProcessed(
        java.lang.String     name,
        java.lang.String     nsPrefix,
        java.lang.String     nsSystemID,
        java.util.Properties extraAttributes,
        java.lang.String     systemID,
        int                  lineNr)
    throws java.lang.Exception;
```

This method is called when the attributes of an XML element have been processed.

If there are attributes with a default value which have not been specified yet, they have to be put in *extraAttributes*.

name	The name of the element.
nsPrefix	The prefix used to identify the namespace
nsSystemID	The system ID associated with the namespace
extraAttributes	Where to put extra attributes
systemID	The system ID of the XML data of the element
lineNr	The line number in the XML data of the element

elementStarted

```
void elementStarted(java.lang.String name,
                    java.lang.String nsPrefix,
                    java.lang.String nsSystemID,
                    java.lang.String systemID,
                    int              lineNr)
    throws java.lang.Exception;
```

Indicates that an element has been started.

name	The name of the element.
nsPrefix	The prefix used to identify the namespace
nsSystemID	The system ID associated with the namespace
systemID	The system ID of the XML data of the element
lineNr	The line number in the XML data of the element

getParameterEntityResolver

```
IXMLEntityResolver getParameterEntityResolver();
```

Returns the parameter entity resolver.

parseDTD

```
void parseDTD(java.lang.String   publicID,
              IXMLReader         reader,
              IXMLEntityResolver resolver,
              boolean            external)
    throws java.lang.Exception;
```

Parses the DTD. The validator object is responsible for reading the full DTD.

publicID	The public ID, which may be null.
reader	The reader to read the DTD from.
resolver	The entity resolver.
external	`true` if the DTD is external

PCDataAdded

```
void PCDataAdded(java.lang.String systemID,
                 int              lineNr)
    throws java.lang.Exception;
```

Indicates that a new #PCDATA element has been encountered.

systemID	The system ID of the XML data of the element
lineNr	The line number in the XML data of the element

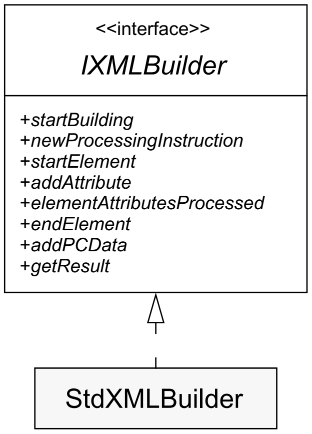
setParameterEntityResolver

```
void setParameterEntityResolver(
        IXMLEntityResolver resolver);
```

Sets the parameter entity resolver.

resolver	The resolver

net.n3.nanoxml.StdXMLBuilder



`StdXMLBuilder` is a concrete implementation of `IXMLBuilder` which creates a tree of `XMLElement` from an XML data source.

addAttribute

```
void addAttribute(java.lang.String key,
                  java.lang.String nsPrefix,
                  java.lang.String nsSystemID,
                  java.lang.String value,
                  java.lang.String type)
    throws Exception;
```

This method is called when a new attribute of an XML element is encountered.

key	The key (name) of the attribute.
nsPrefix	The prefix used to identify the namespace. If no namespace has been specified, this parameter is `null`.
nsSystemID	The system ID associated with the namespace. If no namespace has been specified, or no system ID is associated with nsPrefix, this parameter is `null`.
value	The value of the attribute.
type	The type of the attribute. If no type is known, "CDATA" is returned.

addPCData

```
void addPCData(java.io.Reader   reader,
               java.lang.String systemID,
               int              lineNr)
    throws Exception;
```

This method is called when a PCDATA element is encountered. A Java reader is supplied from which you can read the data. The reader will only read the data of the element. You don’t need to check for boundaries. If you don’t read the full element, the rest of the data is skipped. You also don’t have to care about entities; they are resolved by the parser.

reader	The method can retrieve the parameter of the PI from this reader. You may close the reader before reading all its data and you cannot read too much data.
systemID	The system ID of the XML data source
lineNr	The line in the source where the element starts.

elementAttributesProcessed

```
void elementAttributesProcessed(java.lang.String name,
                                java.lang.String nsPrefix,
                                java.lang.String nsSystemID)
    throws Exception;
```

This method is called when all the attributes of an XML element have been processed.

name	The name of the element.
nsPrefix	The prefix used to identify the namespace. If no namespace has been specified, this parameter is `null`.
nsSystemID	The system ID associated with the namespace. If no namespace has been specified, or no system ID is associated with nsPrefix, this parameter is `null`.

endElement

```
void endElement(java.lang.String name,
                java.lang.String nsPrefix,
                java.lang.String nsSystemID)
    throws Exception;
```

This method is called when the end of an XML element is encountered.

name	The name of the element.
nsPrefix	The prefix used to identify the namespace. If no namespace has been specified, this parameter is `null`.
nsSystemID	The system ID associated with the namespace. If no namespace has been specified, or no system ID is associated with nsPrefix, this parameter is `null`.

getResult

```
java.lang.Object getResult()
    throws Exception;
```

Returns the result of the building process. This method is called just before the *parse* method of `IXMLParser` returns. The result is of type `XMLElement`, but because of limitations of the Java language, I cannot specify that here.

newProcessingInstruction

```
void newProcessingInstruction(java.lang.String target,
                              java.io.Reader   reader)
    throws Exception;
```

This method is called when a processing instruction is encountered. A PI with a reserved target ("xml" with any case) is never reported.

target	The processing instruction target.
reader	The method can retrieve the data from this reader. You may close the reader before reading all its data and you cannot read too much data.

startBuilding

```
void startBuilding(String systemID,
                   int    lineNr)
    throws Exception;
```

This method is called before the parser starts processing its input.

systemID	The system ID of the XML data source
lineNr	The line on which the parsing starts.

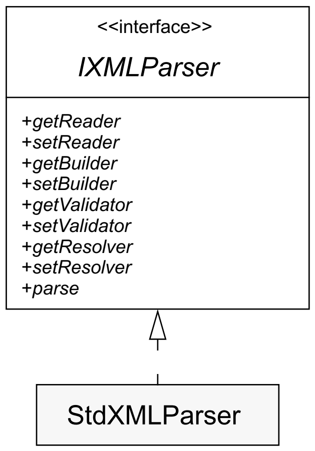
startElement

```
void startElement(java.lang.String name,
                  java.lang.String nsPrefix,
                  java.lang.String nsSystemID,
                  java.lang.String systemID,
                  int              lineNr)
    throws Exception;
```

This method is called when a new XML element is encountered.

name	The name of the element.
nsPrefix	The prefix used to identify the namespace. If no namespace has been specified, this parameter is `null`.
nsSystemID	The system ID associated with the namespace. If no namespace has been specified, or no system ID is associated with nsPrefix, this parameter is `null`.
systemID	The system ID of the XML data source.
lineNr	The line in the source where the element starts.

net.n3.nanoxml.StdXMLParser



`StdXMLParser` is the concrete instance of `IXMLParser`.

getBuilder

```
IXMLBuilder getBuilder();
```

Returns the builder which creates the local structure of the XML data.

getReader

```
IXMLReader getReader();
```

Returns the reader form which the parser retrieves its data.

getResolver

```
IXMLEntityResolver getResolver();
```

Returns the entity resolver.

getValidator

```
IXMLValidator getValidator();
```

Returns the validator that validates the XML data.

parse

```
java.lang.Object parse()
    throws XMLException;
```

Parses the data and lets the builder create the logical structure. The method returns the result of *getData* of the builder described on page 32. If an error occurred while reading or parsing the data, the method may throw an `XMLException`.

setBuilder

```
void setBuilder(IXMLBuilder builder);
```

Sets the builder which creates the local structure of the XML data.

builder	The builder

setReader

```
void setReader(IXMLReader reader);
```

Sets the reader from which the parser retrieves its data.

reader	The reader

setResolver

```
void setResolver(IXMLEntityResolver resolver);
```

Sets the entity resolver.

resolver	The resolver

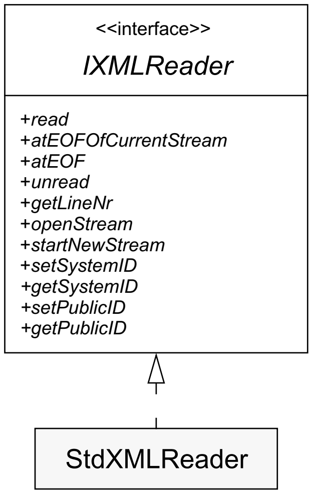
setValidator

```
void setValidator(IXMLValidator validator);
```

Sets the validator that validates the XML data.

validator	The validator

net.n3.nanoxml.StdXMLReader



`STDXMLReader` is the concrete implementation of `IXMLReader` which reads the data to be parsed.

atEOF

```
boolean atEOF()
    throws java.io.IOException;
```

Returns `true` if there are no more characters left to be read.

atEOFOfCurrentStream

```
boolean atEOFOfCurrentStream()
    throws java.io.IOException;
```

Returns `true` if the current stream has no more characters left to be read.

fileReader (static)

```
static IXMLReader fileReader(java.lang.String filename);
```

Creates a new reader using a file as input.

filename	The name of the file containing the XML data.

getEncoding (protected)

```
protected String getEncoding(java.lang.String str);
```

Retrieves the encoding from a `<?xml … ?>` tag. This method is only called when the encoding could not be determined automatically and when the tag is the very first element in the data.

str	The first PI tag in the XML data.

getLineNr

```
int getLineNr();
```

Returns the line number of the data in the current stream.

getPublicID

```
java.lang.String getPublicID();
```

Returns the public ID of the current stream.

getSystemID

```
java.lang.String getSystemID();
```

Returns the system ID of the current stream.

openStream

```
java.io.Reader openStream(java.lang.String publicID,
                          java.lang.String systemID)
        throws java.net.MalformedURLException,
               java.io.FileNotFoundException,
               java.io.IOException;
```

Opens a stream from a public and system ID.

publicID	The public ID of the entity (may be `null`)
systemID	The system ID of the entity.

read

```
char read()
    throws java.io.IOException;
```

Reads a character.

setPublicID

```
void setPublicID(java.lang.String publicID);
```

Sets the public ID of the current stream.

publicID	The public ID.

setSystemID

```
void setSystemID(java.lang.String systemID)
        throws java.net.MalformedURLException;
```

Sets the system ID of the current stream.

systemID	The system ID.

startNewStream

```
void startNewStream(java.io.Reader reader);
```

Starts a new stream from a Java reader. The new stream is used temporary to read data from. If that stream is exhausted, control returns to the "parent" stream.

reader	The reader to read the new data from.

StdXMLReader (constructor)

```
StdXMLReader(java.lang.String publicID,
             java.lang.String systemID)
    throws java.net.MalformedURLException,
           java.io.FileNotFoundException,
           java.io.IOException;
```

Initializes a reader from a system ID and optional public ID.

systemID	The system ID.
publicID	The public ID (may be null)

```
StdXMLReader(java.io.Reader reader);
```

Initializes a reader from a Java reader.

reader	The Java reader.

```
StdXMLReader(java.io.InputStream stream);
```

Initializes a reader from a Java input stream.

stream	The Java input stream.

stream2reader (protected)

```
protected java.io.Reader stream2reader(
        java.io.InputStream    stream,
        java.lang.StringBuffer charsRead);
```

Converts a Java input stream to a Java reader while detecting the encoding.

stream	The input for the XML data.
charsRead	Buffer where to put characters that have been read.

stringReader (static)

```
static IXMLReader stringReader(java.lang.String str);
```

Creates a new reader using a string as input.

str	The string containing the XML data.

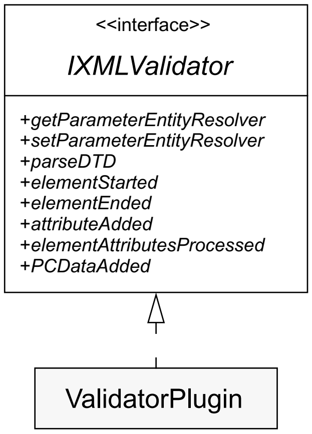
unread

```
void unread(char ch)
        throws java.io.IOException
```

Pushes the last character read back on the stream.

ch	The character to push back

net.n3.nanoxml.ValidatorPlugin



`ValidatorPlugin` allows the application to insert additional validators into the NanoXML framework. More information about custom validators can be found on page 18.

attributeAdded

```
void attributeAdded(java.lang.String key,
                    java.lang.String nsPrefix,
                    java.lang.String nsSystemID,
                    java.lang.String value,
                    java.lang.String systemID,
                    int              lineNr)
    throws java.lang.Exception;
```

Indicates that an attribute has been added.

key	The name of the attribute.
nsPrefix	The prefix used to identify the namespace
nsSystemID	The system ID associated with the namespace
value	The value of the attribute
systemID	The system ID of the XML data of the element
lineNr	The line number in the XML data of the element

elementStarted

```
void elementStarted(java.lang.String name,
                    java.lang.String nsPrefix,
                    java.lang.String nsSystemID,
                    java.lang.String systemID,
                    int              lineNr)
    throws java.lang.Exception;
```

Indicates that an element has been started.

name	The name of the element.
nsPrefix	The prefix used to identify the namespace
nsSystemID	The system ID associated with the namespace
systemID	The system ID of the XML data of the element
lineNr	The line number in the XML data of the element

elementAttributesProcessed

```
void elementAttributesProcessed(
        java.lang.String     name,
        java.lang.String     nsPrefix,
        java.lang.String     nsSystemID,
        java.util.Properties extraAttributes,
        java.lang.String     systemID,
        int                  lineNr)
    throws java.lang.Exception;
```

This method is called when the attributes of an XML element have been processed.

If there are attributes with a default value which have not been specified yet, they have to be put in *extraAttributes*.

name	The name of the element.
nsPrefix	The prefix used to identify the namespace
nsSystemID	The system ID associated with the namespace
extraAttributes	Where to put extra attributes
systemID	The system ID of the XML data of the element
lineNr	The line number in the XML data of the element

getDelegate

```
IXMLValidator getDelegate();
```

Returns the delegate.

getParameterEntityResolver

```
IXMLEntityResolver getParameterEntityResolver();
```

Returns the parameter entity resolver.

invalidAttributeValue

```
void invalidAttributeValue(java.lang.String systemID,
                           int              lineNr,
                           java.lang.String elementName,
                           java.lang.String attributeName,
                           java.lang.String attributeValue)
    throws XMLValidationException;
```

Throws an `XMLValidationException` to indicate that an attribute has an invalid value.

systemID	The system ID of the XML data of the element
lineNr	The line number in the XML data of the element
elementName	The name of the element
attributeName	The name of the attribute
attributeValue	The value of the attribute

missingAttribute

```
void missingAttribute(java.lang.String systemID,
                      int              lineNr,
                      java.lang.String elementName,
                      java.lang.String attributeName)
    throws XMLValidationException;
```

Throws an `XMLValidationException` to indicate that an attribute is missing.

systemID	The system ID of the XML data of the element
lineNr	The line number in the XML data of the element
elementName	The name of the element
attributeName	The name of the missing attribute

missingElement

```
void missingElement(java.lang.String systemID,
                    int              lineNr,
                    java.lang.String parentElementName,
                    java.lang.String missingElementName)
    throws XMLValidationException;
```

Throws an `XMLValidationException` to indicate that an element is missing.

systemID	The system ID of the XML data of the element
lineNr	The line number in the XML data of the element
parentElementName	The name of the parent element
missingElementName	The name of the missing element

missingPCData

```
void missingPCData(java.lang.String systemID,
                   int              lineNr,
                   java.lang.String parentElementName)
    throws XMLValidationException;
```

Throws an `XMLValidationException` to indicate that a `#PCDATA` element was missing.

systemID	The system ID of the XML data of the element
lineNr	The line number in the XML data of the element
parentElementName	The name of the parent element

parseDTD

```
void parseDTD(java.lang.String   publicID,
              IXMLReader         reader,
              IXMLEntityResolver resolver,
              boolean            external)
    throws java.lang.Exception;
```

Parses the DTD. The validator object is responsible for reading the full DTD.

publicID	The public ID, which may be `null`.
reader	The reader to read the DTD from.
resolver	The entity resolver.
external	`true` if the DTD is external

PCDataAdded

```
void PCDataAdded(java.lang.String systemID,
                 int              lineNr)
    throws java.lang.Exception;
```

Indicates that a new #PCDATA element has been encountered.

systemID	The system ID of the XML data of the element
lineNr	The line number in the XML data of the element

setDelegate

```
void setDelegate(IXMLValidator delegate);
```

Sets the delegate.

delegate	The delegate

setParameterEntityResolver

```
void setParameterEntityResolver(
        IXMLEntityResolver resolver);
```

Sets the parameter entity resolver.

resolver	The resolver

unexpectedAttribute

```
void unexpectedAttribute(java.lang.String systemID,
                         int              lineNr,
                         java.lang.String elementName,
                         java.lang.String attributeName)
    throws XMLValidationException;
```

Throws an `XMLValidationException` to indicate that an attribute is unexpected.

systemID	The system ID of the XML data of the element
lineNr	The line number in the XML data of the element
elementName	The name of the element
attributeName	The name of the unexpected attribute

unexpectedElement

```
void unexpectedElement(
        java.lang.String systemID,
        int              lineNr,
        java.lang.String parentElementName,
        java.lang.String unexpectedElementName)
    throws XMLValidationException;
```

Throws an `XMLValidationException` to indicate that an element is unexpected.

systemID	The system ID of the XML data of the element
lineNr	The line number in the XML data of the element
parentElementName	The name of the parent element
unexpectedElementName	The name of the unexpected element

unexpectedPCData

```
void unexpectedPCData(java.lang.String systemID,
                      int              lineNr,
                      java.lang.String parentElementName)
    throws XMLValidationException;
```

Throws an `XMLValidationException` to indicate that a `#PCDATA` element was unexpected.

systemID	The system ID of the XML data of the element
lineNr	The line number in the XML data of the element
parentElementName	The name of the parent element

validationError

```
    void validationError(java.lang.String systemID,
                         int              lineNr,
                         java.lang.String message,
                         java.lang.String elementName,
                         java.lang.String attributeName,
                         java.lang.String attributeValue)
        throws XMLValidationException;
```

Throws a generic `XMLValidationException`.

systemID	The system ID of the XML data of the element
lineNr	The line number in the XML data of the element
message	The error message
elementName	The name of the element (may be `null`)
attributeName	The name of the attribute (may be `null`)
attributeValue	The value of the attribute (may be `null`)

net.n3.nanoxml.XMLElement

addChild

```
    void addChild(XMLElement child);
```

Adds a child element.

child	The child to add.

enumerateAttributeNames

```
    java.util.Enumeration enumerateAttributeNames();
```

Returns an enumeration of all attribute names.

enumerateChildren

```
java.util.Enumeration enumerateChildren();
```

Returns an enumeration of all child elements.

getAttribute

```
java.lang.String getAttribute(java.lang.String name);
```

Returns the value of an attribute. If the attribute is not present, `null` is returned.

name	The name of the attribute.

```
java.lang.String getAttribute(
        java.lang.String name,
        java.lang.String defaultValue);
```

Returns the value of an attribute. If the attribute is not present, *defaultValue* is returned.

name	The name of the attribute.
defaultValue	The default value of the attribute.

getAttributes

```
java.util.Properties getAttributes();
```

Returns a Properties object containing all the properties of the element. You should not modify the result.

getChildAtIndex

```
XMLElement getChildAtIndex(int index);
```

Returns the child element located at a certain index.

index	The index of the child, where the first child has index 0.

getChildren

```
java.util.Vector getChildren();
```

Returns a vector containing all the child elements. You should not modify the vector.

getChildrenCount

```
int getChildrenCount();
```

Returns the number of children.

getChildrenNamed

```
java.util.Vector getChildrenNamed(java.lang.String name);
```

Returns a vector of all child elements named *name*.

name	The name of the children to search.

getContent

```
java.lang.String getContent();
```

Returns the the `#PCDATA` content of the element. If the element has a combination of `#PCDATA` content and child elements, the `#PCDATA` sections can be retrieved as unnamed child objects. In this case, this method returns `null`.

getFirstChildNamed

```
XMLElement getFirstChildNamed(java.lang.String name);
```

Returns the first child named *name*. If there is no such child, `null` is returned.

name	The name of the child to search.

getLineNr

```
int getLineNr();
```

Returns the line number in the data where the element started.

getName

```
java.lang.String getName();
```

Returns the name of the element, or null if the element only contains `#PCDATA`.

getSystemID

```
java.lang.String getSystemID();
```

Returns the system ID of the data where the element started.

hasAttribute

```
boolean hasAttribute(java.lang.String name);
```

Returns **`true`** if the element has an attribute named *name*.

name	The name of the attribute.

hasChildren

```
boolean hasChildren();
```

Returns true if the element has children.

isLeaf

```
boolean isLeaf();
```

Returns true if the element has no children.

removeAttribute

```
void removeAttribute(java.lang.String name);
```

Removes a attribute.

name	The name of the attribute.

removeChild

```
void removeChild(XMLElement child);
```

Removes a child element.

child	The child to add.

removeChildAtIndex

```
void removeChildAtIndex(int index);
```

Removes the child element located at a certain index.

index	The index of the child, where the first child has index 0.

setAttribute

```
void setAttribute(java.lang.String name,
                  java.lang.String value);
```

Sets the value of an attribute.

name	The name of the attribute.
value	The value of the attribute.

setContent

```
void setContent(java.lang.String content);
```

Sets the `#PCDATA` content. It is an error to call this method with a non-null value if there are child objects.

content	The (possible `null`) content.

setName

```
void setName(java.lang.String name);
```

Sets the name of the element.

name	The name of the element.

XMLElement (constructor)

```
XMLElement();
```

Creates an element that can be used for `#PCDATA` content.

```
XMLElement(java.lang.String name);
```

Creates an empty element.

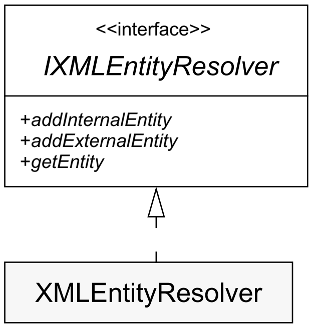
name	The name of the element.

```
XMLElement(java.lang.String name,
           java.lang.String systemID,
           int              lineNr);
```

Creates an empty element.

name	The name of the element.
systemID	The system ID of the XML data where the element starts
lineNr	The line in the XML data where the element starts

net.n3.nanoxml.XMLEntityResolver



An `XMLEntityResolver` resolves entities. More information about custom entity resolvers can be found on page 19.

addExternalEntity

```
void addExternalEntity(java.lang.String name,
                       java.lang.String publicID,
                       java.lang.String systemID);
```

Adds an external entity.

name	The name of the entity.
publicID	The public ID of the entity.
systemID	The system ID of the entity.

addInternalEntity

```
void addInternalEntity(java.lang.String name,
                       java.lang.String value);
```

Adds an internal entity.

name	The name of the entity.
value	The value of the entity.

getEntity

```
java.io.Reader getEntity(IXMLReader       xmlReader,
                         java.lang.String name);
```

Returns a Java reader containing the value of an entity. If the entity could not be resolved, `null` is returned. The method may throw an `XMLParseException` if necessary.

xmlReader	The current NanoXML reader.
name	The name of the entity.

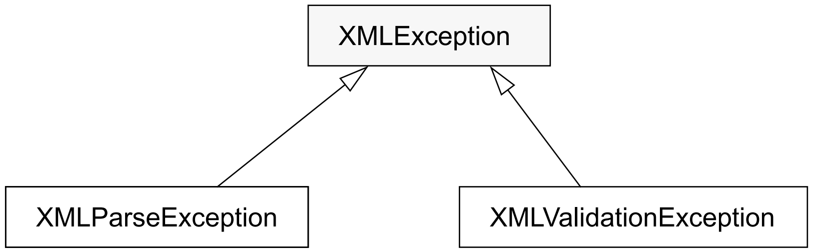
openExternalEntity

```
java.io.Reader openExternalEntity(
        IXMLReader       xmlReader,
        java.lang.String publicID,
        java.lang.String systemID);
```

Opens an external entity. The method may throw an `XMLParseException` if necessary.

xmlReader	The current NanoXML reader.
publicID	The public ID of the entity.
systemID	The system ID of the entity

net.n3.nanoxml.XMLException



An `XMLException` is thrown when an exception occurred while processing the XML data.

getException

```
java.lang.Exception getException();
```

Returns the encapsulated exception, or `null` if no exception is encapsulated.

getLineNr

```
int getLineNr();
```

Returns the line number in the XML data where the exception occurred. If there is no line number known, -1 is returned.

getSystemID

```
java.lang.String getSystemID();
```

Returns the system ID of the XML data where the exception occurred. If there is no system ID known, `null` is returned.

XMLException (constructor)

```
XMLException(java.lang.String msg);
```

Creates an exception.

msg	The message of the exception

```
XMLException(java.lang.Exception exception);
```

Creates an exception.

exception	The encapsulated exception

```
XMLException(java.lang.String systemID,
             int              lineNr,
             java.lang.String msg);
```

Creates an exception.

systemID	The system ID of the XML data where the exception occurred
lineNr	The line number in the XML data where the exception occurred
msg	The message of the exception

```
XMLException(java.lang.String    systemID,
             int                 lineNr,
             java.lang.Exception exception);
```

Creates an exception.

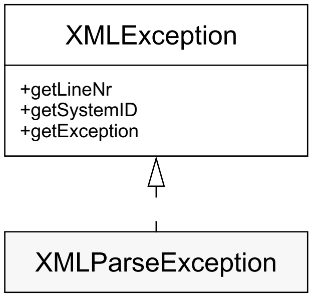
systemID	The system ID of the XML data where the exception occurred
lineNr	The line number in the XML data where the exception occurred
exception	The encapsulated exception

```
XMLException(java.lang.String    systemID,
             int                 lineNr,
             java.lang.Exception exception,
             java.lang.String    msg,
             boolean             reportParams);
```

Creates an exception.

systemID	The system ID of the XML data where the exception occurred
lineNr	The line number in the XML data where the exception occurred
exception	The encapsulated exception
msg	The message of the exception
reportParams	true if the the *systemID*, *lineNr* and *exception* parameters need to be appended to the message

net.n3.nanoxml.XMLParseException



An XMLParseException is thrown when the XML passed to the XML parser is not well-formed.

XMLParseException (constructor)

```
XMLParseException(java.lang.String msg);
```

Creates an exception.

msg	The message of the exception

```
XMLParseException(java.lang.String systemID,
                  int              lineNr,
                  java.lang.String msg);
```

Creates an exception.

systemID	The system ID of the XML data where the exception occurred
lineNr	The line number in the XML data where the exception occurred
msg	The message of the exception

net.n3.nanoxml.XMLParserFactory

Creates an XML parser.

createDefaultParser (static)

```
static IXMLParser createDefaultParser()
    throws java.lang.ClassNotFoundException,
           java.lang.InstantiationException,
           java.lang.IllegalAccessException
```

Creates a default parser. The actual class is dependent on the system property `net.n3.nanoxml.XMLParser`. If this property has not been defined, the default class `net.n3.nanoxml.StdXMLParser` is used.

```
    static IXMLParser createDefaultParser(IXMLBuilder builder)
        throws java.lang.ClassNotFoundException,
               java.lang.InstantiationException,
               java.lang.IllegalAccessException
```

Creates a default parser with a custom builder.

builder	The custom builder

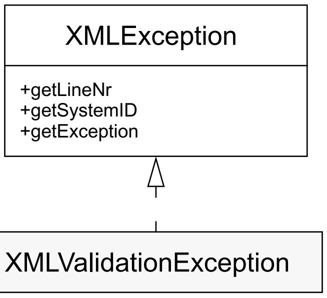
createParser (static)

```
    static IXMLParser createParser(java.lang.String className,
                                   IXMLBuilder      builder)
        throws java.lang.ClassNotFoundException,
               java.lang.InstantiationException,
               java.lang.IllegalAccessException
```

Creates a parser with a custom builder.

className	The name of the class of the parser.
builder	The custom builder

net.n3.nanoxml.XMLValidationException



An XMLValidationException is thrown when the XML passed to the XML parser is well-formed but not valid. There are 8 types of validation errors, each identified by a constant:

MISSING_ELEMENT:
An element was missing

UNEXPECTED_ELEMENT:
An unexpected element was encountered

MISSING_ATTRIBUTE:
An attribute was missing

UNEXPECTED_ATTRIBUTE:
An unexpected attribute was encountered

ATTRIBUTE_WITH_INVALID_VALUE:
An attribute has an invalid value

MISSING_PCDATA:
A `#PCDATA` element was missing

UNEXPECTED_PCDATA:
An unexpected `#PCDATA` element was encountered

MISC_ERROR:
Another error than those specified above was encountered

getAttributeName

```
java.lang.String getAttributeName();
```

Returns the name of the attribute in which the validation is violated. If there is no current attribute, `null` is returned.

getAttributeValue

```
java.lang.String getAttributeValue();
```

Returns the value of the attribute in which the validation is violated. If there is no current attribute, `null` is returned.

getElementName

```
java.lang.String getElementName();
```

Returns the name of the element in which the validation is violated. If there is no current element, `null` is returned.

XMLValidationException (constructor)

```
XMLValidationException(int              errorType,
                       java.lang.String systemID,
                       int              lineNr,
                       java.lang.String elementName,
                       java.lang.String attributeName,
                       java.lang.String attributeValue,
                       java.lang.String msg);
```

Creates a new exception.

errorType	The type of validity error.
systemID	The system ID from where the data came
lineNr	The line number in the XML data where the exception occurred
elementName	The name of the offending element (`null` if n/a)
attributeName	The name of the offending attribute (`null` if n/a)
attributeValue	The value of the offending attribute (`null` if n/a)
msg	The message of the exception

net.n3.nanoxml.XMLWriter

An `XMLWriter` writes XML data to a stream.

write

```
void write(XMLElement xml)
    throws java.io.IOException;
void write(XMLElement xml,
           boolean    prettyPrint);
    throws java.io.IOException
void write(XMLElement xml,
           boolean    prettyPrint,
           int        indent)
    throws java.io.IOException;
```

Writes an XML element.

xml	The XML element to write
prettyPrint	`true` if spaces need to be inserted to make the output more readable (default: `true`)
indent	How many spaces to indent the element (default: `4`; ignored if prettyPrint is `false`)

XMLWriter (constructor)

```
XMLWriter(java.io.Writer output);
XMLWriter(java.io.OutputStream output);
```

Creates a new XML writer.

output	Where to write the output to

Index