NanoXML/Java 2.2

Marc De Scheemaecker <cyberelf@mac.com>

February 1, 2003

Contents

1 Introduction **5**
1.1 About XML 5
1.2 About NanoXML 6
1.3 NanoXML 2 6
1.4 NanoXML Extension to the XML System ID 7

2 Retrieving Data From An XML Datasource **9**
2.1 A Very Simple Example 9
2.2 Analyzing The Data 10
2.3 Generating XML 11
2.4 Namespaces 12

3 Retrieving Data From An XML Stream **15**
3.1 The XML Builder 15
3.2 Registering an XML Builder 17

4 Advanced Topics **19**
4.1 The NanoXML Reader 20
4.2 The NanoXML Parser 21
4.3 The NanoXML Validator 21
4.4 The NanoXML Entity Resolvers 22
4.4.1 Standard Entities 22
4.4.2 Parameter Entities 23
4.5 The NanoXML Builder 23

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.

XMLis 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 April 2002.

1.4 NanoXML Extension to the XML System ID

Because it's convenient to put data files into jar files, we need some way to specify that we want some resource which can be found in the class path. There is no support for such resources in the XML 1.0 specification. NanoXML allows you to specify such resources using the *reference part* of a URL.

This means that if the DTD of the XML data is put in the resource */data/foo.dtd*, you can specify such path using the following document type declaration:

```
<!DOCTYPE foo SYSTEM 'file:#/data/foo.dtd'>
```

It's even possible to specify a resource found in a particular jar, like in the following example:

```
<!DOCTYPE foo SYSTEM 'http://myserver.com/dtds.jar#/foo.dtd'>
```

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.

2.1 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 = XMLParserFactory.createDefaultXMLParser();  ②
    IXMLReader reader = StdXMLReader.fileReader("test.xml");        ③
    parser.setReader(reader);
    IXMLElement xml = (IXMLElement) parser.parse();                 ④
    XMLWriter writer = new XMLWriter(System.out);                   ⑤
    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 XMLdata on an output stream. In this case, we dump the read data to the standard output (System.out).

2.2 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 `IXMLElement` 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 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 `getFullName`, thus:

```
FOO.getFullName() → "FOO"
```

You can enumerate the attribute keys using `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", null) → "fred"
```

The child elements can be enumerated using `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 XMLelements as its children, the character data segments will be put in untitled XMLelements (whose name is null).

`IXMLElement` contains many convenience methods for retrieving data and traversing the XMLtree.

2.3 Generating XML

You can very easily create a tree of XMLelements or modify an existing one.

To create a new tree, just create an `IXMLElement` object:

```
IXMLElement 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`:

```
IXMLElement child = elt.createElement("Child");
elt.addChild(child);
```

Note that the child element is created calling `createElement`. This insures that the child instance is compatible with its new parent.

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, which you create by calling `createPCDataElement`:

```
IXMLElement pcdata = elt.createPCDataElement();
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 = ...;
IXMLElement xmltree = ...;
XMLWriter xmlwriter = new XMLWriter(output);
writer.write(xmltree);
```

2.4 Namespaces

As of version 2.1, NanoXML has support for namespaces. Namespaces allow you to attach a URI to the name of an element name or an attribute. This URI allows you to make a distinction between similary named entities coming from different sources. More information about namespaces can be found in the XML Namespaces recommendation, which can be found at http://www.w3c.org/TR/REC-xml-names/.

Please note that a DTD has no support for namespaces. It is important to understand that an XMLdocument can have only one DTD. Though the namespace URI is often presented as a URL, that URL is not a system id for a DTD. The only function of a namespace URI is to provide a globally unique name.

As an example, lets have the following XMLdata:

```
<doc:book xmlns:doc="http://nanoxml.n3.net/book">
  <chapter xmlns="http://nanoxml.n3.net/chapter"
           title="Introduction"
           doc:id="chapter1"/>
</doc:book>
```

The top-level element uses the namespace “http://nanoxml.n3.net/book”. The prefix is used as an alias for the namespace, which is defined in the attribute xmlns:doc. This prefix is defined for the doc:book element and its child elements.

The chapter element uses the namespace “http://nanoxml.n3.net/chapter”. Because the namespace URI has been defined as the value of the xmlns attribute, the namespace is the default namespace for the chapter element. Default namespaces are inherited by the child elements, but only for their names. Attributes never have a default namespace.

The chapter element has an attribute doc:id, which is defined in the same namespace as doc:book because of the doc prefix.

NanoXML 2.1 offers some variants on the standard retrieval methods to allow the application to access the namespace information.

In the following examples, we assume the variable book to contain the doc:book element and the variable chapter to contain the chapter element.

To get the full name, which includes the namespace prefix, of the element, use `getFullName`:

```
book.getFullName() → "doc:book"
chapter.getFullName() → "chapter"
```

To get the short name, which excludes the namespace prefix, of the element, use `getName`:

```
book.getName() → "book"
chapter.getName → "chapter"
```

For elements that have no associated namespace, `getName` and `getFullName` are equivalent.

To get the namespace URI associated with the name of the element, use `getNamespace`:

```
book.getNamespace() → "http://nanoxml.n3.net/book"
chapter.getNamespace() → "http://nanoxml.n3.net/chapter"
```

If no namespace is associated with the name of the element, this method returns `null`.

You can get an attribute of an element using either its full name (which includes its prefix) or its short name together with its namespace URI, so the following two instructions are equivalent:

```
chapter.getAttribute("doc:id", null)
chapter.getAttribute("id", "http://nanoxml.n3.net/book", null)
```

Note that the title attribute of chapter has no namespace, even though the chapter element name has a default namespace.

You can create a new element which uses a namespace this way:

```
book = new XMLElement("doc:book", "http://nanoxml.n3.net/book");
chapter = book.createElement("chapter",
                             "http://nanoxml.n3.net/chapter");
```

You can add an attribute which uses a namespace this way:

```
chapter.setAttribute("doc:id",
                     "http://nanoxml.n3.net/book",
                     chapterId);
```

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 until it's completely read, you can use streaming.

3.1 The XML Builder

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

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 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(String name,        ⑥
                                         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 stored in `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.

3.2 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.

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

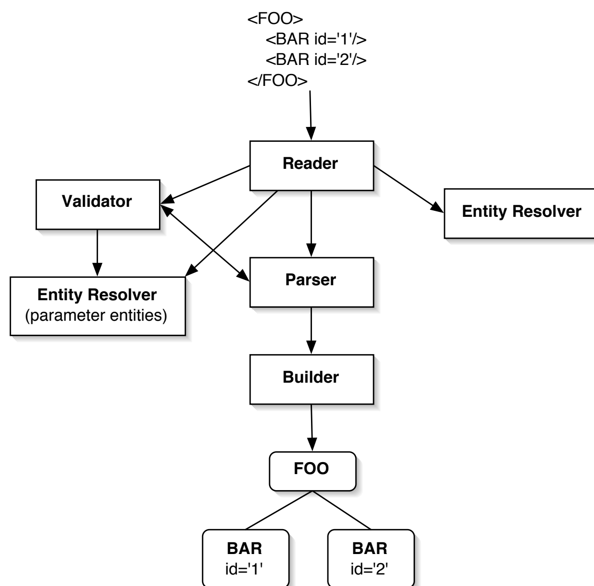

Figure 4.1: Design of NanoXML/Java

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.

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 `IXMLElement`. 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.

4.1 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.

4.2 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`.

4.3 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.

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 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, 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);
```

4.4 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.

4.4.1 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);
    IXMLElement xml = (IXMLElement) parser.parse();
    XMLWriter writer = new XMLWriter(System.out);
    writer.write(xml);
  }
}
```

4.4.2 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);
    IXMLElement xml = (IXMLElement) parser.parse();
    XMLWriter writer = new XMLWriter(System.out);
    writer.write(xml);
  }
}
```

4.5 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>
```