XOM Tutorial

Elliotte Rusty Harold

Table of Contents

XOM is designed to be easy to learn and easy to use. It works very straight-forwardly, and has a very shallow learning curve. Assuming you're already familiar with XML, you should be able to get up and running with XOM very quickly.

Creating XML Documents

Let's begin, as customary, with a Hello World program. In particular, suppose we want to create this XML document:

```
<?xml version="1.0?>
<root>
  Hello World!
</root>
```

First we have to import the `nu.xom` package where most of the interesting classes live:

```
import nu.xom.*;
```

This document contains a single element, named root, so we create an `Element` object named "root":

```
Element root = new Element("root");
```

Next we append the string `"Hello World!"` to it:

```
root.appendChild("Hello World!");
```

Now that we have the root element, we can use it to create the `Document` object:

```
Document doc = new Document(root);
```

We can create a `String` containing the XML for this `Document` object using its `toXML` method:

```
String result = doc.toXML();
```

This string can be written onto an `OutputStream` or a `Writer` in the usual way. Here's the complete program:

Example 1. Hello World with XOM

```
import nu.xom.*;

public class HelloWorld {

  public static void main(String[] args) {

    Element root = new Element("root");
    root.appendChild("Hello World!");
    Document doc = new Document(root);
    String result = doc.toXML();
    System.out.println(result);

  }
```

```
}
```

This is compiled and run in the usual way. When that's done, here's the output:

```
<?xml version="1.0"?>
<root>Hello World!</root>
```

You may notice that this isn't quite what the goal was. The white space is different. On reflection, this shouldn't be too surprising. White space is significant in XML. If you want line breaks and indentation, you should include that in the strings you use to construct the data. For example,

```
root.appendChild("\n  Hello World!\n");
```

Appending children

Let's write a more complicated document. In particular, let's write a document that encodes the Fibonacci numbers in XML, like this:

```
<?xml version="1.0"?>
<Fibonacci_Numbers>
  <fibonacci>1</fibonacci>
  <fibonacci>1</fibonacci>
  <fibonacci>2</fibonacci>
  <fibonacci>3</fibonacci>
  <fibonacci>5</fibonacci>
  <fibonacci>8</fibonacci>
  <fibonacci>13</fibonacci>
  <fibonacci>21</fibonacci>
  <fibonacci>34</fibonacci>
  <fibonacci>55</fibonacci>
</Fibonacci_Numbers>
```

Begin by creating the root `Fibonacci_Numbers` element:

```
Element root = new Element("Fibonacci_Numbers");
```

Next we need a loop that creates the individual `fibonacci` elements. After it's created each of these elements is appended to the root element using the `appendChild` method:

```
for (int i = 1; i <= 10; i++) {
  Element fibonacci = new Element("fibonacci");
  fibonacci.appendChild(low.toString());
  root.appendChild(fibonacci);

  BigInteger temp = high;
  high = high.add(low);
```

```
  low = temp;
}
```

Next we create the document from the root element, and print it on `System.out`:

```
  Document doc = new Document(root);
  System.out.println(doc.toXML());
```

Here's the completed program:

Example 2. Generating Fibonacci Numbers in XML

```
import java.math.BigInteger;
import nu.xom.*;

public class FibonacciXML {

  public static void main(String[] args) {

      BigInteger low  = BigInteger.ONE;
      BigInteger high = BigInteger.ONE;

      Element root = new Element("Fibonacci_Numbers");
      for (int i = 1; i <= 10; i++) {
        Element fibonacci = new Element("fibonacci");
        fibonacci.appendChild(low.toString());
        root.appendChild(fibonacci);

        BigInteger temp = high;
        high = high.add(low);
        low = temp;
      }
      Document doc = new Document(root);
      System.out.println(doc.toXML());

  }

}
```

This is compiled and run in the usual way. When that's done, here's the output:

```
<?xml version="1.0"?>
<Fibonacci_Numbers><fibonacci>1</fibonacci><fibonacci>1</fibonacci><fib
```

Serializer

Once again the white space isn't quite what we wanted. This is a good opportunity to introduce the `Serializer` class. Instead of using `toXML`, you can ask a `Serializer` object to write the document onto an `OutputStream`. You can also tell the `Serializer` to insert line breaks and

indents in reasonable places. For instance, Example 3 requests a four space indent, the ISO-8859-1 (Latin-1) encoding, and a 64 character maximum line length:

Example 3. Using a Serializer to Output XML

```
import nu.xom.*;

import java.io.IOException;
import java.math.BigInteger;

public class PrettyFibonacci {

  public static void main(String[] args) {

    BigInteger low  = BigInteger.ONE;
    BigInteger high = BigInteger.ONE;

    Element root = new Element("Fibonacci_Numbers");
    for (int i = 1; i <= 10; i++) {
        Element fibonacci = new Element("fibonacci");
        fibonacci.appendChild(low.toString());
        root.appendChild(fibonacci);

        BigInteger temp = high;
        high = high.add(low);
        low = temp;
    }
    Document doc = new Document(root);

    try {
      Serializer serializer = new Serializer(System.out, "ISO-8859-1");
      serializer.setIndent(4);
      serializer.setMaxLength(64);
      serializer.write(doc);
    }
    catch (IOException ex) {
       System.err.println(ex);
    }

  }

}
```

Here's the output, much more nicely formatted:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Fibonacci_Numbers>
    <fibonacci>1</fibonacci>
    <fibonacci>1</fibonacci>
    <fibonacci>2</fibonacci>
    <fibonacci>3</fibonacci>
    <fibonacci>5</fibonacci>
    <fibonacci>8</fibonacci>
    <fibonacci>13</fibonacci>
    <fibonacci>21</fibonacci>
    <fibonacci>34</fibonacci>
    <fibonacci>55</fibonacci>
</Fibonacci_Numbers>
```

Besides, line length and indentation, `Serializer` gives you several other options for controlling the output including:

- The line separator string (\r\n by default)
- The character encoding (UTF-8 by default)
- Whether to insert `xml:base` attributes to retain the base URI property
- Whether to normalize output using Unicode normalization form C

There are a few things you should note about using a `Serializer`:

- By default, `Serializer` outputs an XML document that precisely represents a XOM `Document`. If you parse the serialized output back in to XOM, you'll get an exactly equivalent tree. [1] All the text content of the document that is part of a document's infoset is precisely preserved. This includes boundary white space. Insignificant white space such as white space inside tags is not included in the XML information set, and generally will not be preserved.
- If you tell `Serializer` to change a document's infoset by inserting line breaks and/or indenting, it may trim, compress, or remove existing white space as well. It does not limit itself merely to adding white space.
- `Serializer` makes reasonable efforts to respect the requested maximum line length and indentation, but it does not guarantee that it will do so. For instance, if an element name is 50 characters long and the maximum line length is 40, then `Serializer` will generate a line longer than 40 characters.
- No matter what options are set, `Serializer` does not change white space in elements where `xml:space="preserve"`.
- If the `Serializer` cannot output a character in the current encoding, it will try to escape it with a numeric character reference. If it cannot use a numeric character reference (for

instance, because the unavailable character occurs in an element name), it throws an `UnavailableCharacterException`. This is a runtime exception. This should not happen in UTF-8 and UTF-16 encodings.

Attributes

Adding attributes is not hard. In XOM, the `Attribute` class represents attributes, and it works pretty much as you'd expect. For example, this statement creates an `Attribute` object representing the attribute `id="p1"`:

```
Attribute a = new Attribute("id", "p1");
```

The `addAttribute` method in the `Element` class attaches an attribute to an `Element` object. If there's an existing attribute with the same local name and namespace URI, it's removed at the same time. Example 4 demonstrates with a simple program that adds some `index` attributes to the `fibonacci` elements:

Example 4. Adding attributes to elements

```
import java.math.BigInteger;
import nu.xom.*;

public class AttributeFibonacci {

  public static void main(String[] args) {

      BigInteger low  = BigInteger.ONE;
      BigInteger high = BigInteger.ONE;

      Element root = new Element("Fibonacci_Numbers");
      for (int i = 1; i <= 10; i++) {
        Element fibonacci = new Element("fibonacci");
        fibonacci.appendChild(low.toString());
        Attribute index = new Attribute("index", String.valueOf(i));
        fibonacci.addAttribute(index);
        root.appendChild(fibonacci);

        BigInteger temp = high;
        high = high.add(low);
        low = temp;
      }
      Document doc = new Document(root);
      System.out.println(doc.toXML());

  }

}
```

When this program is run, it produces the following output (after adding a few line breaks):

```
<?xml version="1.0"?>
<Fibonacci_Numbers xmlns=""><fibonacci index="1">1</fibonacci><fibonacc
<fibonacci index="3">2</fibonacci><fibonacci index="4">3</fibonacci>
<fibonacci index="5">5</fibonacci><fibonacci index="6">8</fibonacci>
<fibonacci index="7">13</fibonacci><fibonacci index="8">21</fibonacci>
<fibonacci index="9">34</fibonacci><fibonacci index="10">55</fibonacci>
```

Document Type Declarations

Suppose you have a DTD sitting at the relative URL `fibonacci.dtd`. Example 5 creates a document type declaration pointing to that DTD, and then attaches it to the document:

Example 5. Including a document type declaration

```
import nu.xom.*;
import java.math.BigInteger;

public class ValidFibonacci {

  public static void main(String[] args) {

      BigInteger low  = BigInteger.ONE;
      BigInteger high = BigInteger.ONE;

      Element root = new Element("Fibonacci_Numbers");
      for (int i = 1; i <= 10; i++) {
        Element fibonacci = new Element("fibonacci");
        fibonacci.appendChild(low.toString());
        Attribute index = new Attribute("index", String.valueOf(i));
        fibonacci.addAttribute(index);
        root.appendChild(fibonacci);

        BigInteger temp = high;
        high = high.add(low);
        low = temp;
      }
      Document doc = new Document(root);
      DocType doctype = new DocType("Fibonacci_Numbers", "fibonacci.dtd
      doc.insertChild(doctype, 0);
      System.out.println(doc.toXML());

  }

}
```

One thing XOM does not allow you to do is create an internal DTD subset. You can parse one from an input document, and it will be preserved in the document type declaration as the document is manipulated, but you cannot create a new one. The reason is that XOM is fanatical about maintaining well-formedness, and XOM

cannot currently check the well-formedness of DTD declarations. It has to rely on the parser to do that.

Note

If you really need the internal DTD subset, you can create a string containing a document with the internal DTD subset you want, parse that string to forma `Document` object, detach the temporary document's `DocType` object, and add that to another document. For example,

```
Element greeting = new Element("greeting");
Document doc = new Document(greeting);
String temp = "<!DOCTYPE element [\n"
  + "<!ELEMENT greeting (#PCDATA)\n"
  + "]>\n"
  + "<root />";
Builder builder = new Builder();
Document tempDoc = builder.build(temp, null);
DocType doctype = tempDoc.getDocType();
doctype.detach();
doc.setDocType(doctype);
```

Namespaces

XOM fully supports namespaces, and enforces all namespace constraints. It does not allow developers to create namespace malformed documents. You can create elements, attributes, and documents that don't use namespaces at all. However, if you do use namespaces you have to follow the rules. In fact, XOM is actually a little more strict than the namespaces spec technically requires. It insists that all namespace URIs be syntactically correct, absolute URIs according to RFC 2396. The main effect is that you can't use non-ASCII characters such as γ and Ω in namespace URIs. These must all be properly percent escaped before passing them to XOM.

That said, XOM's namespace model is possibly the cleanest of all the major APIs. It has two basic rules you need to remember:

- If an element or attribute has a prefix, use the qualified name when constructing the object or changing the name.
- The qualified name is always the first argument and the

namespace URI is always the second argument to any method that takes both. Namespace URIs are just strings, so it is possible to inadvertently swap the arguments, but don't worry: if you get them backwards, XOM throws an exception that alerts you to your mistake very quickly. [2]

For example, this code fragment creates a `p` element in no namespace:

```
Element paragraph = new Element("p");
```

To place the element in the XHTML namespace, just add a second argument containing the XHTML namespace URI:

```
Element paragraph = new Element("p", "http://www.w3.org/TR/2001/xhtml")
```

To make the element prefixed, just add the prefix to the name:

```
Element paragraph = new Element("html:p", "http://www.w3.org/TR/2001/xh
```

Example 6 demonstrates with a simple program that outputs the Fibonacci numbers as a MathML document:

Example 6. Creating elements in namespaces

```
import nu.xom.*;
import java.math.BigInteger;
import java.io.IOException;

public class MathMLFibonacci {

  public static void main(String[] args) {

      BigInteger low  = BigInteger.ONE;
      BigInteger high = BigInteger.ONE;

      String namespace = "http://www.w3.org/1998/Math/MathML";
      Element root = new Element("mathml:math", namespace);
      for (int i = 1; i <= 10; i++) {
        Element mrow = new Element("mathml:mrow", namespace);
        Element mi = new Element("mathml:mi", namespace);
        Element mo = new Element("mathml:mo", namespace);
        Element mn = new Element("mathml:mn", namespace);
        mrow.appendChild(mi);
        mrow.appendChild(mo);
        mrow.appendChild(mn);
        root.appendChild(mrow);
        mi.appendChild("f(" + i + ")");
        mo.appendChild("=");
        mn.appendChild(low.toString());

        BigInteger temp = high;
        high = high.add(low);
```

```
        low = temp;
      }
      Document doc = new Document(root);

      try {
        Serializer serializer = new Serializer(System.out, "ISO-8859-1"
        serializer.setIndent(4);
        serializer.setMaxLength(64);
        serializer.write(doc);
      }
      catch (IOException ex) {
        System.err.println(ex);
      }
  }

}
```

Here's the output:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<mathml:math xmlns:mathml="http://www.w3.org/1998/Math/MathML">
    <mathml:mrow>
        <mathml:mi>f(1)</mathml:mi>
        <mathml:mo>=</mathml:mo>
        <mathml:mn>1</mathml:mn>
    </mathml:mrow>
    <mathml:mrow>
        <mathml:mi>f(2)</mathml:mi>
        <mathml:mo>=</mathml:mo>
        <mathml:mn>1</mathml:mn>
    </mathml:mrow>
    <mathml:mrow>
        <mathml:mi>f(3)</mathml:mi>
        <mathml:mo>=</mathml:mo>
        <mathml:mn>2</mathml:mn>
    </mathml:mrow>
    <mathml:mrow>
        <mathml:mi>f(4)</mathml:mi>
        <mathml:mo>=</mathml:mo>
        <mathml:mn>3</mathml:mn>
    </mathml:mrow>
    <mathml:mrow>
        <mathml:mi>f(5)</mathml:mi>
        <mathml:mo>=</mathml:mo>
        <mathml:mn>5</mathml:mn>
    </mathml:mrow>
    <mathml:mrow>
        <mathml:mi>f(6)</mathml:mi>
        <mathml:mo>=</mathml:mo>
        <mathml:mn>8</mathml:mn>
    </mathml:mrow>
    <mathml:mrow>
        <mathml:mi>f(7)</mathml:mi>
        <mathml:mo>=</mathml:mo>
        <mathml:mn>13</mathml:mn>
    </mathml:mrow>
    <mathml:mrow>
        <mathml:mi>f(8)</mathml:mi>
        <mathml:mo>=</mathml:mo>
        <mathml:mn>21</mathml:mn>
    </mathml:mrow>
    <mathml:mrow>
        <mathml:mi>f(9)</mathml:mi>
```

```
        <mathml:mo>=</mathml:mo>
        <mathml:mn>34</mathml:mn>
    </mathml:mrow>
    <mathml:mrow>
        <mathml:mi>f(10)</mathml:mi>
        <mathml:mo>=</mathml:mo>
        <mathml:mn>55</mathml:mn>
    </mathml:mrow>
</mathml:math>
```

You never have to worry about adding `xmlns` and `xmlns:`*`prefix`* attributes. XOM always handles that for you automatically. Indeed if you try to create attributes with these names, XOM will throw an `IllegalNameException` . Sometimes, however, namespace prefixes are used in element content and attribute values, even though those prefixes aren't used on any names anywhere in the document. This is common in XSLT, for example. In this case, you may have to add extra namespace declarations to certain elements to bind these prefixes to the correct URI. This is done with `Element`'s `addNamespaceDeclaration` method. For example, this code fragment binds the prefix `svg` to the namespace URI `http://www.w3.org/TR/2000/svg`:

```
element.addNamespaceDeclaration("svg", "http://www.w3.org/TR/2000/svg")
```

This technique can also be used to force common namespace declarations onto the root element when serializing.

Parsing XML Documents

Much of the time, of course, you don't create the original document in XOM. Instead, you read an existing XML document from a file, a network socket, a URL, a `java.io.Reader`, or some other input source. The `Builder` class is responsible for reading a document and constructing a XOM `Document` object from it. For example, this attempts to read the document at http://www.cafeconleche.org/:

```
try {
  Builder parser = new Builder();
  Document doc = parser.build("http://www.cafeconleche.org/");
}
catch (ParsingException ex) {
  System.err.println("Cafe con Leche is malformed today. How embarrassi
}
catch (IOException ex) {
  System.err.println("Could not connect to Cafe con Leche. The site may
}
```

You'll notice that the `build` method may throw a `ParsingException` if the document is malformed or namespace malformed. It may also throw a `java.io.IOException` if the document cannot be read. Both of these are checked exceptions that must be declared or caught.

Depending on platform, relative URLs may or may not be interpreted as file names. On Windows they seem to be. On Unix/Linux, they are not. It is much safer to use full, unrelative file URLs such as file:///home/elharo/Projects/data/example.xml which should work on essentially any platform. Alternately, you can pass a `java.io.File` object to the `build` method instead of a URL. You can also pass an `InputStream` or a `Reader` from which the XML document will be read.

You can also build a `Document` from a `String` that contains the actual XML document. In this case, you must provide a second argument giving the base URL of the document, which would otherwise not be available. For example,

```
Document doc = parser.build("<greeting>Hello World!</greeting>", "http:
```

If there really is no base URL, you can pass null for the second argument. However, this will prevent the resolution of any relative URLs within the document, and may prevent the document from being parsed if the document type declaration uses a relative URL.

Validating

By default XOM only checks for well-formedness and namespace well-formedness. If you want it to check for validity too (and throw a `ValidityException` if a violation is detected) you can pass `true` to the `Builder` constructor, like this:

```
try {
  Builder parser = new Builder(true);
  Document doc = parser.build("http://www.cafeconleche.org/");
}
catch (ValidityException ex) {
  System.err.println("Cafe con Leche is invalid today. (Somewhat embarr
}
catch (ParsingException ex) {
  System.err.println("Cafe con Leche is malformed today. (How embarrass
}
catch (IOException ex) {
  System.err.println("Could not connect to Cafe con Leche. The site may
}
```

A `ValidityException` is not fatal. The entire document is parsed anyway. If you still want to process the invalid document, you can invoke the `getDocument` method of `ValidityException` to return a `Document` object. For example,

```
Document doc;
try {
  Builder parser = new Builder(true);
  doc = parser.build("http://www.cafeconleche.org/");
}
catch (ValidityException ex) {
  doc = ex.getDocument();
}
catch (ParsingException ex) {
  System.err.println("Cafe con Leche is malformed today. (How embarrass
  System.exit(1);
}
catch (IOException ex) {
  System.err.println("Could not connect to Cafe con Leche. The site may
  System.exit(1);
}
```

`ValidityException` also contains methods you can use to list the validity errors in the document:

```
public int getErrorCount()
public String getValidityError(int n)
```

The exact number of exceptions and the content of the error messages depends on the underlying parser.

Setting SAX Properties

If you need to control the specific parser class used, you can create a SAX `XMLReader` in the usual way, and then pass it to the `Builder` constructor. For instance, this would allow you to use John Cowan's TagSoup to parse an HTML document into XOM:

```
  try {
    XMLReader tagsoup = XMLReaderFactory.createXMLReader("org.ccil.cowa
    Builder bob = new Builder(tagsoup);
    Document yahoo = bob.build("http://www.yahoo.com");
    // ...
  }
  catch (SAXException ex) {
    System.out.println("Could not load Xerces.");
    System.out.println(ex.getMessage());
  }
```

You can configure a SAX parser before passing it to XOM. For example, suppose you want to use Xerces to perform schema

validation. You would set up the `Builder` thusly:

```
String url = "http://www.example.com/";
try {
  XMLReader xerces = XMLReaderFactory.createXMLReader("org.apache.xer
  xerces.setFeature("http://apache.org/xml/features/validation/schema

  Builder parser = new Builder(xerces, true);
  parser.build(url);
  System.out.println(url + " is schema valid.");
}
catch (SAXException ex) {
  System.out.println("Could not load Xerces.");
  System.out.println(ex.getMessage());
}
catch (ParsingException ex) {
  System.out.println(args[0] + " is not schema valid.");
  System.out.println(ex.getMessage());
  System.out.println(" at line " + ex.getLineNumber()
    + ", column " + ex.getColumnNumber());
}
catch (IOException ex) {
  System.out.println("Due to an IOException, Xerces could not check "
}
```

This mechanism is primarily intended for custom SAX properties and features such as schema validation or filters. XOM requires certain standard SAX properties to be set in certain ways: In particular, XOM expects to control the following parser properties and features:

- `http://xml.org/sax/features/namespace-prefixes`
- `http://xml.org/sax/features/external-general-entities`
- `http://xml.org/sax/features/external-parameter-entities`
- `http://xml.org/sax/features/namespace-prefixes`
- `http://xml.org/sax/features/validation`
- `http://xml.org/sax/features/string-interning`
- `http://apache.org/xml/features/allow-java-encodings`
- `http://apache.org/xml/features/standard-uri-conformant`
- `http://xml.org/sax/properties/lexical-handler`
- `http://xml.org/sax/properties/declaration-handler`

Any values you provide for these properties and features will be

overridden by XOM when it constructs the `Builder`. Similarly, `Builder` expects to be able to set all handlers: `ContentHandler`, `DeclHandler`, `ErrorHandler`, etc. If you hang onto a reference to the `XMLReader`, you could probably change them back later; but don't do that. If you do XOM will get very confused, and probably break sooner rather than later.

Navigation

Once you have a document in memory, you're going to want to navigate it. The primary navigation methods are declared in the `Node` class so they're accessible on everything in the tree.

```
public final Document getDocument()
public final ParentNode getParent()
public abstract int getChildCount()
public final Node getChild(int i)
```

The normal strategy in XOM is a `for` loop that iterates across the children, often recursing down the tree. The first child is at position 0. The last child is at one less than the number of children of the node. For example,

```
    public static void process(Node node) {

        // Do whatever you're going to do with this node…

        // recurse the children
        for (int i = 0; i < node.getChildCount(); i++) {
            process(node.getChild(i));
        }

    }
```

Example 7 shows a simple program that recursively descends through a document, printing out an indented view of the nodes it spots on the way. It uses the `getChild` and `getChildCount` methods as well as the `getRootElement` from the `Document` class.

Example 7. A program that prints all the nodes in a document

```
import java.io.*;
import nu.xom.*;

public class NodeLister {
```

```
  public static void main(String[] args) {

    if (args.length == 0) {
      System.out.println("Usage: java nu.xom.samples.NodeLister URL");
      return;
    }

    Builder builder = new Builder();

    try {
      Document doc = builder.build(args[0]);
      Element root = doc.getRootElement();
      listChildren(root, 0);
    }
    // indicates a well-formedness error
    catch (ParsingException ex) {
      System.out.println(args[0] + " is not well-formed.");
      System.out.println(ex.getMessage());
    }
    catch (IOException ex) {
      System.out.println(ex);
    }

  }

  public static void listChildren(Node current, int depth) {

    printSpaces(depth);
    String data = "";
    if (current instanceof Element) {
        Element temp = (Element) current;
        data = ": " + temp.getQualifiedName();
    }
    else if (current instanceof ProcessingInstruction) {
        ProcessingInstruction temp = (ProcessingInstruction) current;
        data = ": " + temp.getTarget();
    }
    else if (current instanceof DocType) {
        DocType temp = (DocType) current;
        data = ": " + temp.getRootElementName();
    }
    else if (current instanceof Text || current instanceof Comment) {
        String value = current.getValue();
        value = value.replace('\n', ' ').trim();
        if (value.length() <= 20) data = ": " + value;
        else data = ": " + current.getValue().substring(0, 17) + "...";
    }
    // Attributes are never returned by getChild()
    System.out.println(current.getClass().getName() + data);
    for (int i = 0; i < current.getChildCount(); i++) {
      listChildren(current.getChild(i), depth+1);
    }

  }

  private static void printSpaces(int n) {

    for (int i = 0; i < n; i++) {
      System.out.print(' ');
    }

  }

}
```

For example, here's the beginning of output when I ran this program against Cafe con Leche:

```
$ java -classpath .:xom-1.0b3.jar NodeLister http://www.cafeconleche.or
nu.xom.Element: html
 nu.xom.Text:
 nu.xom.Element: head
  nu.xom.Text:
  nu.xom.Element: title
   nu.xom.Text: Cafe con Leche XM...
  nu.xom.Text:
  nu.xom.Element: meta
  nu.xom.Text:
  nu.xom.Element: meta
  nu.xom.Text:
  nu.xom.Element: link
  nu.xom.Text:
  nu.xom.Element: link
  nu.xom.Text:
  nu.xom.Element: meta
  nu.xom.Text:
  nu.xom.Element: script
   nu.xom.Text:
   nu.xom.Comment:
/* Only sunsites...
```

Top-down descent is the primary navigation path most XOM programs take, and the one for which XOM is most optimized.

Element Navigation

In addition, if all you care about are the elements, then the `Element` class includes several methods that allow you to navigate exclusively by element, while ignoring other nodes. You can filter elements by local name and namespace. Passing null for the name argument returns all elements in the specified namespace.

```
public final Elements getChildElements()
public final Elements getChildElements(String name)
public final Elements getChildElements(String name,
String namespaceURI)
```

You'll notice these three methods all return an `Elements` object. This is a type-safe, read-only list that only contains elements. It has two methods, `get` and `size`:

```
public Element get(int index)
public int size()
```

Like most lists in Java, the first element is at position 0 and the last is at one less than the length of the list. For example, this method recursively lists all the elements in an element:

```
public static void listChildren(Element current, int depth) {
  System.out.println(current.getQualifiedName());
  Elements children = current.getChildElements();
   for (int i = 0; i < children.size(); i++) {
    listChildren(children.get(i), depth+1);
  }

}
```

Sometimes, of course, you don't want a list of all the child elements. You just want one. For this purpose, XOM has the `getFirstChildElement` methods:

```
public final Element getFirstChildElement(String name)
public final Element getFirstChildElement(String name,
String namespaceURI)
```

These are mostly useful when you really expect there won't be more than one such child, and you don't want the extra hassle of list iteration. The name is intended to convey the fact that even if you expect that there is only one such child, there may in fact be more. In any case, the first one is always returned. If there's no child with the necessary name and namespace URI, then these methods return null.

Example 8 uses these methods to find the title of any well-formed web page, the assumption being that the page has only one of those. First it looks for a `title` element in no namespace. If that fails it looks for a `title` element in the XHTML namespace.

Example 8. A program to find the title of a web page

```
import nu.xom.*;
import java.io.IOException;

public class TitleSearch {

  public static void main(String[] args) {

   if (args.length == 0) {
     System.err.println("Usage: java TitleSearch url");
     return;
   }

   String pageURL = args[0];

   Builder builder = new Builder();
```

```
    try {
      Document doc = builder.build(pageURL);
      Element html = doc.getRootElement();
      Element head = html.getFirstChildElement("head");
      if (head == null) {
        head = html.getFirstChildElement("head", "http://www.w3.org/1999
      }
      Element title = head.getFirstChildElement("title");
      if (title == null) {
        title = head.getFirstChildElement("title", "http://www.w3.org/19
      }
      System.out.println(title.getValue());
    }
    catch (NullPointerException ex) {
      System.err.println(pageURL + " does not have a title.");
    }
    catch (ParsingException ex) {
      System.err.println(pageURL + " is malformed.");
    }
    catch (IOException ex) {
      System.err.println("Could not read " + pageURL);
    }

  }

}
```

Here's the output when run on Cafe con Leche:

```
$ java -classpath .:../../build/xom-1.0b3.jar TitleSearch http://www.ca
Cafe con Leche XML News and Resources
```

Siblings

XOM does not include any methods for direct access to siblings. You can find a node's previous or next sibling by getting the node's position within its parent node and then adding or subtracting one. This is accomplished with the `indexOf` method in the `ParentNode` class.

```
public int indexOf(Node child)
```

For example, this method finds the next sibling of any specified node, or returns null, if the node is the last child of its parent or does not have a parent:

```
public static Node getNextSibling(Node current) {
  ParentNode parent = current.getParent();
  if (parent == null) return null;
  int index = parent.indexOf(current);
  if (index+1 == parent.getChildCount()) return null;
  return parent.getChild(index+1);
}
```

A slight variant of this operation allows you to navigate through an

entire document along what XPath would call the following axis:

```
public static Node getNext(Node current) {
  ParentNode parent = current.getParent();
  if (parent == null) return null;
  int index = parent.indexOf(current);
  if (index+1 == parent.getChildCount()) return getNext(parent);
  return parent.getChild(index+1);
}
```

However, `indexOf` is a relatively expensive operation, especially for broad nodes with lots of children. `getNextSibling` is a lot faster in many DOM implementations. However, the cost is carrying around an extra pointer inside each node. At an extra four bytes per object, this adds up fast. In most cases, you can design your processing so you navigate through the tree in order, asking for each child of the parent in turn without using `indexOf`.

Attributes

The `Element` class provides six methods to inquire about the attributes of an element:

- You can iterate over all the element's attributes using `getAttribute(int i)` and `getAttributeCount`.

  ```
  public final int getAttributeCount()
  public final Attribute getAttribute(int index)
  ```

  The order of the attributes in this list is unpredictable, not necessarily reproducible, and may not match the order of the attributes in the original document. Namespace declarations (`xmlns` and `xmlns:foo` attributes) are not included in this list.

- You can ask for a specific attribute by its name or its local name and namespace URI:

  ```
  public final Attribute getAttribute(String name)
  public final Attribute getAttribute(String localName,
  String namespaceURI)
  ```

  These two methods return null if no such attribute exists.

- You can also ask for the value of a specific attribute by its

name or its local name and namespace URI:

```
public final String getAttributeValue(String name)
public final String getAttributeValue(String localName,
String namespaceURI)
```

These two methods also return null if no such attribute exists.

For example, suppose we wanted to allow Example 7 to also print attributes. We could rewrite the first branch in the `listChildren` method like so:

```
if (current instanceof Element) {
    Element temp = (Element) current;
    data = ": " + temp.getQualifiedName();
    for (int i = 0; i < temp.getAttributeCount(); i++) {
      Attribute attribute = temp.getAttribute(i);
      String attValue = attribute.getValue();
      attValue = attValue.replace('\n', ' ').trim();
      if (value.length() >= 20) {
        attValue = attValue.substring(0, 17) + "...";
      }
      data += "\r\n    "
      data += attribute.getQualifiedName();
      data += "="
      data += attValue();
    }
}
```

The Node Superclass

In the XOM data model, there are seven types of object found in an XML document:

- `Document`
- `Element`
- `DocType`
- `Text`
- `Comment`
- `ProcessingInstruction`
- `Attribute`

All of these are direct or indirect subclasses of `Node`. `Node` defines the basic methods all XOM node objects support, including methods to:

Get the parent of this node:

```
public final ParentNode getParent()
```

This method returns null if the node does not currently have a parent. XOM never allows a node to have more than one parent at a time, though a node can be removed from one parent and added to another.

Get the document that contains this node:

```
public final Document getDocument()
```

This method returns null if the node does not currently belong to a document. XOM never allows a node to belong to more than one document at a time, though nodes can be moved from one document to another.

Calculate the XPath 1.0 string-value of a node:

```
public abstract String getValue()
```

The XPath rules for calculating string-values that XOM follows are:

- The value of a text node is the text of the node.
- The value of a comment is the text of the comment.
- The value of a processing instruction is the processing instruction data, but does not include the target.
- The value of an element is the concatenation of the values of all the text nodes contained within that element, in document order.
- The value of a document is the value of the root element of the document.
- The value of an attribute is the normalized value of the attribute. (If the attribute is created in memory, the value is the exact text of the attribute as specified. No extra

normalization is performed. However, if the attribute is serialized white space is escaped as necessary to prevent serialization.)

XPath doesn't define a string-value for document type declarations, so XOM returns the empty string as the value of all `DocType` nodes.

This method never returns null, though it may return the empty string.

Get the base URI of a node:

```
public String getBaseURI()
```

Base URIs are calculated according to the XML Base Specification and RFC 2396, taking account of both `xml:base` attributes and the original URIs of the entities from which the node was parsed. In the cases of nodes created in memory with no obvious base URI, this method returns the empty string. The base URI is always an absolute URI, or the empty string if an absolute URI cannot be formed from the information in the document and the object.

Remove a node from its parent:

```
public void detach()
```

After a node has been detached, it may be inserted in another parent, in the same or a different document.

Get the children of a node:

```
public abstract int getChildCount()
public final Node getChild(int i)
```

Theoretically, these three methods really shouldn't be in this class because not all nodes have children. Logically, they belong to the `ParentNode` class. However, in practice it turns out to be very useful to ask a node for its children without knowing whether it can have any. Therefore for leaf nodes such as text nodes and processing instructions, `getChildCount` returns 0, and `getChild` throws an `IndexOutOfBoundsException`.

`Node` also defines a couple of general utility methods:

Get the XML representation of a node:

```
public abstract String toXML()
```

This method returns the actual `String` form of the XML representing this node. Invoking `toXML` on a `Document` is often simpler than setting up a full `Serializer` if you don't need to set formatting options like indenting and maximum white space. However, since this builds the entire document in memory, it can be problematic for large documents and less efficient than using a `Serializer`, which can stream the document. For small documents, the difference rarely matters.

Copy a node:

```
public abstract Node copy()
```

This is a deep copy. However, the return value has no parent and is not part of any document.

The `Node` class also overrides the `equals` and `hashCode` methods. Equality between nodes is defined as identity. That is, two nodes are equal if and only if they are the same object. XOM depends on this definition of equality internally, so both `equals` and `hashCode` are declared final, and cannot be overridden in subclasses.

The ParentNode Class

A parent node is a node that can contain other nodes. In the XOM data model, there are two types of parent nodes, `Document` and `Element`. In XOM, a parent node does not contain a list of children. Rather it *is* a list. Like most lists in Java, these begin at 0 and continue to one less than the length of the list (the number of children the parent has). The `ParentNode` class has methods for appending, inserting, removing, finding, and replacing child nodes:

```
public void insertChild(Node child, int position)
public void appendChild(Node child)
public int indexOf(Node child)
public Node removeChild(Node child)
public void replaceChild(Node oldChild, Node newChild)
public Node removeChild(int position)
```

These methods all enforce the usual well-formedness constraints. For example, if you try to insert a `Text` into a `Document` or a `DocType` into an `Element`, an `IllegalAddException` is thrown. If you try to insert a child beyond the bounds of the parent, an `IndexOutOfBoundsException` is thrown. These are all runtime exceptions so you don't need to explicitly catch them unless you expect something to go wrong.

Because XML Base only defines base URIs in terms of elements and documents (i.e., the base URI of a non-parent node is the base URI of its parent), this class also contains the `setBaseURI` method:

```
public void setBaseURI(String URI)
```

Factories, Filters, Subclassing, and Streaming

XOM is designed for subclassing. You can write your own subclasses of the standard XOM node classes that provide special methods or enforce additional constraints. For instance an HTML XOM could include classes for `P`, `Div`, `Table`, `Head`, and so forth, all subclasses of `Element`.

To support subclasses, the `Builder` does not invoke constructors in the node classes directly. Instead it uses a `NodeFactory`, summarized in Example 9. You can replace the `Builder`'s standard `NodeFactory` with a subclass of your own that creates instances of your subclasses instead of the standard XOM classes.

Example 9. The NodeFactory class

```
package nu.xom;

public class NodeFactory  {

  public NodeFactory();

  public Element  makeRootElement();
  public Element  startMakingElement(String name, String namespace);
  public Nodes    finishMakingElement(Element element);
  public Document startMakingDocument();
  public void     finishMakingDocument(Document document);
  public Nodes    makeAttribute(String name, String uri, String value,
  public Nodes    makeText(String text);
  public Nodes    makeComment(String text);
  public Nodes    makeProcessingInstruction(String target, String data)
  public Nodes    makeDocType(String rootElementName, String publicID,
```

```
}
```

For example, let's suppose you want to add `getInnerXML()` and `setInnerXML()` methods to the `Element` class that enable you to encode XML directly in `String` literals like this:

```
element.setInnerXML(
  "<p>Here's some text</p>\r\n<p>Here's some <em>more</em> text</p>");
```

I am undecided about whether such a method is a good idea or not, but let's allow it for the moment for the sake of argument, or at least the example. To enable this, first you write a subclass of `Element` that adds the extra methods. One such is shown in Example 10.

Example 10. The InnerElement class

```
package nu.xom.samples.inner;

import java.io.IOException;

import nu.xom.*;

public class InnerElement extends Element {


    private static ThreadLocal builders = new ThreadLocal() {

         protected synchronized Object initialValue() {
             return new Builder(new InnerFactory());
         }

     };


    public InnerElement(String name) {
        super(name);
    }


    public InnerElement(String namespace, String name) {
        super(namespace, name);
    }


    public InnerElement(Element element) {
        super(element);
    }


    public String getInnerXML() {

        StringBuffer sb = new StringBuffer();
        for (int i = 0; i < getChildCount(); i++) {
            sb.append(getChild(i).toXML());
        }
        return sb.toString();
```

```
    }

    public void setInnerXML(String xml) throws ParsingException {

        xml = "<fakeRoot>"
          + xml + "</fakeRoot>";
        Document doc;
        try {
            doc = ((Builder) builders.get()).build(xml, null);
        }
        catch (IOException ex) {
            throw new ParsingException(ex.getMessage(), ex);
        }
        this.removeChildren();
        Nodes children = doc.getRootElement().removeChildren();
        for (int i = 0; i < children.size(); i++) {
            this.appendChild(children.get(i));
        }

    }

    public Node copy() {
        return new InnerElement(this);
    }

}
```

Note that when subclassing `Element` you'll want to override the `copy()` as well as any other methods you choose to override.

It's easy enough to create such `InnerElement` objects using constructors; but how to make the `Builder` create them when parsing a document? Simple. Create a `NodeFactory` that returns these elements instead of instances of the base `Element` class and then install it with the `Builder` before parsing. Example 11 shows such a factory class. It overrides `startMakingElement()`. A factory that used custom classes for attributes, comments, processing instructions, and so forth would override additional methods as well. However, this factory does not so it can simply inherit all those other methods.

Example 11. The InnerFactory class that creates InnerElement objects

```
package nu.xom.samples.inner;

import nu.xom.*;

public class InnerFactory extends NodeFactory {

    public Element startMakingElement(String namespaceURI, String name)
        return new InnerElement(namespaceURI, name);
    }
```

```
}
```

Finally you create an instance of the factory and pass it to the `Builder` constructor like so:

```
  private Builder builder = new Builder(new InnerFactory());
  Document doc = builder.build("<root><a>test</a><b>test2</b></root>",
  InnerElement root = (InnerElement) doc.getRootElement();
```

The only inconvenience is that you will need to cast the elements to `InnerElement` in order to use its extra methods. A class that merely overrode existing methods but did not add any new ones would not need to do this.

Node factories are not limited to returning a representation of the item that was actually seen in the document. They can change this item in a variety of ways. As well as removing it completely, they can replace it with a different item, or with several items. They can change a name or a namespace. They can add or remove attributes from an element. The only restriction is that well-formedness must be maintained. For instance, the `makeComment` method can't return a `Text` object if the comment was in the document prolog.

However, you'll note that most of the `NodeFactory` methods are not declared to return the obvious type. For instance, `makeComment` doesn't return a `Comment`, and `makeProcessingInstruction` doesn't return a `ProcessingInstruction`. Instead they both return `Nodes` objects.

`Nodes` is a type-safe, read-write list that can hold any XOM `Node` object. This class provides the usual list methods for getting, removing, and inserting nodes in the list, as well as querying the size of the list and constructors for creating new `Nodes` lists. Example 12 summarizes this class.

Example 12. The Nodes class

```
package nu.xom;

public class Nodes  {

  public Nodes();
  public Nodes(Node initialMember);

  public int  size();
  public Node get(int index);
  public Node remove(int index);
  public void insert(Node node, int index);
```

```
  public void append(Node node);

}
```

Because the factory methods return `Nodes` objects instead of the more specific type, factories can play tricks like converting all comments to elements or replacing one element with several different elements. This flexibility enables a `NodeFactory` to act as a very powerful filter. For instance, one of the simpler filters you can write is one that saves memory by pruning the document tree of the leaves you aren't interested in by returning empty lists. If you know you're going to ignore all processing instructions, a `makeProcessingInstruction` method can simply return an empty `Nodes`. Then `ProcessingInstruction` objects will never even be created. They won't take up any memory, and no time will expended creating them. Similarly you can eliminate all comments by returning an empty `Nodes` from `makeComment`. You can eliminate all attributes by returning an empty `Nodes` from `makeAttribute`, and so forth. Example 13 demonstrates a simple `NodeFactory` that throws away the document type declaration and all comments and processing instructions, so you're only left with the real information content of the document:

Example 13. A Node Factory that strips out the document type declaration, comments and processing instructions

```
import nu.xom.*;

public class JunkStripper extends NodeFactory {

    private Nodes empty = new Nodes();

    public Nodes makeComment(String data) {
        return empty;
    }

    public Nodes makeProcessingInstruction(String target, String data)
        return empty;
    }

    public Nodes makeDocType(String rootElementName,
      String publicID, String systemID) {
        return empty;
    }

}
```

Filters can change data as well as removing it. Example 14 demonstrates a class that encodes all text, comments, processing instructions, and attribute values by ROT13 encoding them.

Example 14. A Node Factory that ROT13 encodes all text

```
import java.io.*;
import nu.xom.*

public class StreamingROT13 extends NodeFactory {

    public static String rot13(String s) {

        StringBuffer out = new StringBuffer(s.length());
        for (int i = 0; i < s.length(); i++) {
            int c = s.charAt(i);
            if (c >= 'A' && c <= 'M') out.append((char) (c+13));
            else if (c >= 'N' && c <= 'Z') out.append((char) (c-13));
            else if (c >= 'a' && c <= 'm') out.append((char) (c+13));
            else if (c >= 'n' && c <= 'z') out.append((char) (c-13));
            else out.append((char) c);
        }
        return out.toString();

    }

    public Nodes makeComment(String data) {
        return new Nodes(new Comment(rot13(data)));
    }

    public Nodes makeText(String data) {
        return new Nodes(new Text(rot13(data)));
    }

    public Nodes makeAttribute(String name, String namespace,
      String value, Attribute.Type type) {
        return new Nodes(new Attribute(name, namespace, rot13(value), t
    }

    public Nodes makeProcessingInstruction(
      String target, String data) {
        return new Nodes(new ProcessingInstruction(rot13(target), rot13
    }

    public static void main(String[] args) {

        if (args.length <= 0) {
          System.out.println("Usage: java nu.xom.samples.StreamingROT13
          return;
        }

        try {
          Builder parser = new Builder(new StreamingROT13());

          // Read the document
          Document document = parser.build(args[0]);

          // Write it out again
          Serializer serializer = new Serializer(System.out);
          serializer.write(document);

        }
        catch (IOException ex) {
          System.out.println(
          "Due to an IOException, the parser could not encode " + args[
          );
        }
        catch (ParsingException ex) {
```

```
            System.out.println(ex);
            ex.printStackTrace();
          }

      } // end main

}
```

Elements are more complex. They have both a beginning and an end. When the `Builder` calls `startMakingElement`, the element has not yet been created. You can either create the `Element` object here and return it, or you can return null. If you return null, then the element's start-tag and end-tag will be omitted from the finished tree, but the element's children will still be included. If you want to replace or remove the element completely, you need to wait for the `Builder` to call the `finishMakingElement` method. At this time, the element has been completely constructed and all its children are in place. You can either return a `Nodes` object containing the `Element` itself, or you can return a `Nodes` list containing other nodes. Whichever you return will be added to the finished tree.

Overriding `finishMakingElement` is an extremely powerful technique that enables XOM to process documents larger than available memory. The trick is to do your processing inside the `NodeFactory` rather than waiting until the entire document has been built. This is typically useful in long documents that consist of very many repetitions of one element; for instance a stock ticker or a data acquisition system. The key element(s) would be processed inside the `finishMakingElement` method. Often this is done in isolation without considering anything outside that element. Once you're finished processing the element, return an empty `Nodes` from `finishMakingElement`. The element will be removed from the tree, and becomes available for garbage collection.

Example 15 demonstrates this technique with a simple program that prints out all the element names in an XML document.

Example 15. A Node Factory that lists elements names

```
import nu.xom.*;
import java.io.IOException;

public class StreamingElementLister extends NodeFactory{

  private int depth = 0;
  private Nodes empty = new Nodes();
```

```
public static void main(String[] args) {

  if (args.length == 0) {
      System.out.println(
        "Usage: java nu.xom.samples.StreamingElementLister URL"
      );
      return;
  }

  Builder builder = new Builder(new StreamingElementLister());

  try {
      builder.build(args[0]);
  }
  catch (ParsingException ex) {
      System.out.println(args[0] + " is not well-formed.");
      System.out.println(ex.getMessage());
  }
  catch (IOException ex) {
      System.out.println(ex);
  }

}

// We don't need the comments.
public Nodes makeComment(String data) {
  return empty;
}

// We don't need text nodes at all
public Nodes makeText(String data) {
  return empty;
}

public Element startMakingElement(String name, String namespace) {
  depth++;
  printSpaces();
  System.out.println(name);
  return new Element(name, namespace);
}

public Nodes finishMakingElement(Element element) {
  depth--;
  if (element.getParent() instanceof Document) {
      return new Nodes(element);
  }
  else return empty;
}

public Nodes makeAttribute(String name, String URI,
  String value, Attribute.Type type) {
    return empty;
}

public Nodes makeDocType(String rootElementName,
  String publicID, String systemID) {
    return empty;
}

public Nodes makeProcessingInstruction(
  String target, String data) {
    return empty;
}
```

```
  private void printSpaces() {
    for (int i = 0; i <= depth; i++) {
      System.out.print(' ');
    }
  }

}
```

In general functionality, this is quite similar to the program we wrote earlier in Example 7. However, they're a couple of crucial differences:

1. This program begins producing output almost immediately. It does not have to wait for the entire document to be parsed.

2. It can process arbitrarily large documents. It is not limited by the available memory.

You don't always need these characteristics in a program; but when you do, XOM makes them really easy to achieve.

One final note on this subject: so far all the examples have treated all elements equally. However, that's absolutely not required. There's no reason you can't key your processing off of the element's name, namespace, attributes, child elements, or other characteristics. For instance, you could remove all XHTML elements from a document or remove all elements except XHTML elements. To invoke the default processing for an element you don't want to filter or modify, just call `super.finishMakingElement(element)`. This is an extremely flexible and powerful technique for processing XML.

XPath

XOM 1.1 and later support XPath queries on nodes. This is often a more robust reliable, and easier way to query a document than explicitly navigating its tree. For example, to find the `title` elements in a Docbook 4 document, you can simply type:

```
Nodes titles = document.query("//title");
```

The `query` method returns a list of nodes, not a single `Node` object. This list may contain zero, one, or more than one `title` elements, the exact number depending solely on what's in the document being

queried. Again, this is in keeping with the design of XPath. The DTD or schema may require that each document have exactly one `title` element; but that doesn't mean this is in fact the case. XPath queries documents as they are, not as they're supposed to be.

Next suppose you need to find the `title` elements in an XHTML document. DocBook 4 doesn't have a namespace, but XHTML does. This requires you to set up an `XPathContext` to bind the prefixes used in the XPath expression to URIs.

```
XPathContext context = new XPathContext("html", "http://www.w3.org/1999
Nodes titles = document.query("//html:title", context);
```

The namespace prefixes in the XPath expression are not necessarily the same ones used in the `Document` object or the document itself. In this case, even though the XHTML documents uses the default namespace, XPath queries must use prefixed names like `html:title` rather than unprefixed names like `title`. This is a basic principle of XPath, and indeed of Namespaces in XML. Only the URI matters. The prefix is just a placeholder.

XSLT

XOM can load an XSLT stylesheet from a XOM `Document` and apply it to another XOM `Document` object. The class that does this is `nu.xom.xslt.XSLTransform`. Each `XSLTransform` object is configured with a particular stylesheet. Then you can apply this stylesheet to other XOM `Document` objects using the `transform` method. For example, this code fragment transforms a document and prints the result on `System.out`.

```
Builder builder = new Builder();
try {
  Document input = builder.build("http://www.example.com/input.xml");
  Document stylesheet = builder.build("http://www.example.com/styleshee
  XSLTransform transform = new XSLTransform(stylesheet);
  Nodes output = transform.transform(input);
  for (int i = 0; i < output.size(); i++) {
    System.out.print(output.get(i).toXML());
  }
  System.out.println();
}
catch (XSLException ex) {
  System.err.println("XSLT error");
}
catch (ParsingException ex) {
  System.err.println("Well-formedness error in " + ex.getURI());
```

```
}
catch (IOException ex) {
  System.err.println("I/O error while reading input document or stylesh
}
```

The result of a transformation is a XOM `Nodes` object. The `Nodes` list returned by the `transform` method may contain zero, one, or more than one node, depending on what the stylesheet produced. After all, there's no guarantee that an XSL transformation produces a well-formed XML document. Sometimes it only produces a well-balanced document fragment, and sometimes it produces nothing at all. However, many stylesheets do produce well-formed XML documents. `XSLTransform` includes a static `toDocument` utility method that converts a `Nodes` object into a `Document` object. However, if the `Nodes` passed to this method contains no elements, more than one element, or any `Text` objects, then `toDocument` throws an `XMLException`. For example,

```
Builder builder = new Builder();
try {
  Document input = builder.build("http://www.example.com/input.xml");
  Document stylesheet = builder.build("http://www.example.com/styleshee
  XSLTransform transform = new XSLTransform(stylesheet);
  Nodes output = transform.transform(input);
  Document result = XSLTransform.toDocument(output);
  System.out.println(result.toXML());
}
catch (XMLException ex) {
  System.err.println("Result did not contain a single root.");
}
catch (XSLException ex) {
  System.err.println("Stylesheet error");
}
catch (ParsingException ex) {
  System.err.println("Well-formedness error in " + ex.getURI());
}
catch (IOException ex) {
  System.err.println("I/O error while reading input document or stylesh
}
```

Because the result of a transformation is a XOM `Nodes` object, not a serialized XML document, any `xsl:output` elements in the stylesheet have no effect on the result of the transformation.

Custom Node Factories

You can provide a `NodeFactory` to be used for building the result tree. This allows you to transform into instances of particular subclasses of the standard XOM classes, rather than the normal classes such as `nu.xom.Element` and `nu.xom.Text`. For example,

```
NodeFactory factory = new CustomNodeFactory();
Builder builder = new Builder();
try {
  Document stylesheet = builder.build("http://www.example.com/styleshee
  XSLTransform transform = new XSLTransform(stylesheet, factory);
  //...
}
catch (XSLException ex) {
  System.err.println("XSLT error");
}
catch (ParsingException ex) {
  System.err.println("Well-formedness error in " + ex.getURI());
}
catch (IOException ex) {
  System.err.println("I/O error while reading input document or stylesh
}
```

Canonicalization

The `nu.xom.canonical.Canonicalizer` class can serialize a XOM document as canonical XML. It is used much like a `Serializer`. For example, this code fragment writes the canonical form of Cafe con Leche onto `System.out`:

```
Builder builder = new Builder();
Canonicalizer outputter = new Canonicalizer(System.out);
Document input = builder.build("http://www.cafeconleche.org/");
outputter.write(input);
```

When canonicalizing you do not have any options to choose the line break character, indentation, maximum line length, encoding, or configure the output in any other way. The purpose of canonical XML is to serialize the same document in a byte-for-byte predictable and reproducible fashion.

XInclude

XOM supports XInclude including the XPointer `element()` scheme and bare name XPointers. It does not support the XPointer `xpointer()` scheme. While internally the XInclude code is one of the ugliest parts of XOM, externally it is extremely simple. You merely pass a `Document` object to the static `XIncluder.resolve()` method, and you get back a new `Document` object in which all `xi:include` elements have been replaced by the content they refer to. The original `Document` object is not changed. For example,

```
Document input = builder.build(url);
```

```
Document result = XIncluder.resolve(input);
```

If something should go wrong during the inclusion process, either an `IOException`, an `XIncludeException`, or one of its subclasses is thrown as appropriate. For example, if a `xi:include` element were to attempt to include itself, either directly or indirectly, an `InclusionLoopException` would be thrown.

You have the option to specify a `Builder` to be used for including. This would allow you to validate the included documents or install a custom `NodeFactory` that returned instances of particular subclasses. For example, this code fragment throws a `ValidityException` if the master document or any of the documents it includes, directly or indirectly, are invalid:

```
try {
  Builder builder = new Builder(true);
  Document input = builder.build("http://www.example.org/master.xml");
  Document result = XIncluder.resolve(input, builder);
}
catch (ValidityException ex) {
  System.err.println("Validity error in " + ex.getURI());
}
```

Summary

This has been a fairly quick tour of XOM. If this tutorial didn't show you how to do what you need to do, try looking in the JavaDoc or the `nu.xom.samples` package. If you still can't figure out how to do what you need to do, you can ask the xom-interest mailing list. I monitor it pretty closely, so most questions are responded to quickly. I prefer you to ask question about XOM on the list rather than e-mailing me personally, since if you have a question, chances are others do too. You do not need to subscribe to post. However, non-subscribers posts are moderated, so for the fastest response you may wish to subscribe.

---

[1] There's one minor possible difference. Depending on where you stored the output, the base URIs of some nodes may not be the same.

[2] This is the advantage of requiring that namespace names be absolute URIs. Most absolute URIs are not legal element names and vice versa so XOM notices if the arguments are swapped.