dom4j cookbook

Mr. Tobias Rademacher

Mr. James Strachan

dom4j cookbook

by Mr. Tobias Rademacher and Mr. James Strachan

Published September 2001

This document provides a practical introduction to dom4j. It guides you through by using a lot of examples and is based on dom4j v1.0

Table of Contents

Foreword

Chapter 1. Introducing dom4j

dom4j is a object model representing an XML Tree in memory. dom4j offers a easy-to-use API that provides a powerfull set of features to process, manipulate or navigate XML and work with XPath and XSLT as well as integrate with SAX, JAXP and DOM.

dom4j is designed to be interface-based in order to provide highly configurable implementation strategies. You are able to create your own XML tree implementations by simply providing a DocumentFactory implementation. This makes it very simple to resuse much of the dom4j code while extending it to provide whatever implementation features you wish.

This document will guide you through dom4j's features in a pratical way. It uses a lot of examples with source code to achive that. The document is also desinged as a reference so that you don't have to read the entire document right now. This guide concentrates on daily work with dom4j and is therefore called *cookbook*.

Chapter 2. Creation of an XML Object Model using dom4j

Normally it all starts with a set of xml-files or a single xml file that you want to process, manipulate or navigate through to extract some values necessary in your application. Most Java Open-Source projects using XML for deployment or as a replacement for property files in order to get easily readable property data.

Reading XML data

How does dom4j help you to get at the data stored in XML? dom4j comes with a set of builder classes that parses the xml data and creating a tree like object structure in memory. You can easily manipulate and navigate through that model. The following example shows how you can read your data using dom4j API.

```
import java.io.File;
import org.dom4j.Document;
import org.dom4j.DocumentException;
import org.dom4j.io.SAXReader;

public class DeployFileLoaderSample {

  /** dom4j object model representation of a xml document. Note: We u
  private Document doc;

  /**
   * Loads a document from a file.
   *
   * @throw a org.dom4j.DocumentException occurs whenever the buildpr
   */
  public void parseWithSAX(File aFile) throws DocumentException {
    SAXReader xmlReader = new SAXReader();
    this.doc = xmlReader.read(aFile);
  }
}
```

The above example code should clarify the use of `org.dom4j.io.SAXReader` to build a complete dom4j-tree from a given file. The org.dom4j.io package of dom4j contains a set of classes for creating and serializing XML objects. The read() method is overloaded so that you able to pass different kind of object that represents a source.

- `java.lang.String` - a SystemId is a String that contains a URI e.g. a URL to a XML file
- `java.net.URL` - represents a Uniform Ressource Loader or a Uniform

Ressource Identifier encasulate in a URL instance

- `java.io.InputStream` - a open input stream that transports xml data
- `java.io.Reader` - more compartable puls the abiltiy of setting the encoding scheme
- `org.sax.InputSource` - a single input source for a XML entity.

So we decide to add more flexiblity to our `DeployFileLoaderSample` and add new methods.

```
import java.io.File;

import org.dom4j.Document;
import org.dom4j.DocumentException;
import org.dom4j.io.SAXReader;

public class DeployFileLoaderSample {

  /** dom4j object model representation of a xml document. Note: We u
  private Document doc;

  /**
   * Loads a document from a file.
   *
   * @param aFile the data source
   * @throw a org.dom4j.DocumentExcepiton occurs whenever the buildpr
   */
  public void parseWithSAX(File aFile) throws DocumentException {
    SAXReader xmlReader = new SAXReader();
    this.doc = xmlReader.read(aFile);
  }

  /**
   * Loads a document from a file.
   *
   * @param aURL the data source
   * @throw a org.dom4j.DocumentExcepiton occurs whenever the buildpr
   */
  public void parseWithSAX(URL aURL) throws DocumentException {
    SAXReader xmlReader = new SAXReader();
    this.doc = xmlReader.read(aURL);
  }


}
```

Integrating with other XML APIs

dom4j offers also classes for integration with the two original XML processing APIs - SAX and DOM. So far we have been talking about reading a document with SAX. The `org.dom4j.SAXContentHandler` class implements several SAX interfaces directly (such as ContentHandler) so that you can embed dom4j directly inside any SAX application. You can also use this class to implement your own specific SAX-based Reader class if you need to.

The `DOMReader` class allows you to convert an existing DOM tree into a dom4j tree. This could be usefull if you already used DOM and want to replace it step by step with dom4j or if you just needs some of DOM's behaviour and want to save memory ressources by transforming it in a dom4j Model. You are able to transform a DOM Document, a DOM node branch and a single element.

```
import org.sax.Document;

import org.dom4j.Document;
import org.dom4j.DocumentException;
import org.dom4j.io.DOMReader;

public class DOMIntegratorSample {

  /** converts a W3C DOM document into a dom4j document */
  public Document buildDocment(org.w3c.dom.Document domDocument) {
    DOMReader xmlReader = new DOMReader();
    return xmlReader.read(domDocument);
  }
}
```

The secret of DocumentFactory

Right now we have talked a lot of reading exisiting XML information e.g. from files, URL's or even Streams. Sometimes it's necessary to generate a XML document from scratch within a running Java Application. The class `org.dom4j.DocumentFactory`defines a set of factory methods to create documents, document types, elements, attributes, unparsed character data (CDATA), a namespace, an XPathobject, a NodeFilter and some other useful instances. This makes the `DocumentFactory`class to a central class whenever you have to create one of these instances by yourself.

```
import org.dom4j.DocumentFactory;
import org.dom4j.Document;
import org.dom4j.Element;

public class DeployFileCreator {

  private DocumentFactory factory = DocumentFactory.getInstance();
  private Document doc;

  public void generateDoc(String aRootElement) {
    doc = DocumentFactory.getInstance().createDocument();
    Element root = doc.addElement(aRootElement);
  }

}
```

The listing shows how to generate a new Document from scratch. The method `generateDoc(String aRootElement)` takes a String parameter. The string value contains the name of the root element of the new document. As you can see `org.dom4j.DocumentFactory` is a singleton that is accessable via `getInstance()` as most Java singletons are. After we obtained the instance we can `DocumentFactory` methods. They follow the *createXXX()* naming convention, so if you want to create a Attribute you would call *createAttribute()* instead. If your class uses DocumentFactory a lot or uses a different DocumentFactory instance then you could add it as a member variable and initiate it via getInstance in your constructor.

```
import org.dom4j.DocumentFactory;
import org.dom4j.Document;
import org.dom4j.Element;

public class GranuatedDeployFileCreator {

 private DocumentFactory factory;
 private Document doc;

 public GranuatedDeployFileCreator() {
   this.factory = DocumentFactory.getInstance();
 }

 public void generateDoc(String aRootElement) {
    doc = factory.createDocument();
    Element root = doc.addElement(aRootElement);
 }
```

```
}
```

The `Document` and `Element` interfaces have a number of helper methods for creating an XML document programmatically in a simple way.

```
import org.dom4j.Document;
import org.dom4j.DocumentHelper;
import org.dom4j.Element;

public class Foo {

  public Document createDocument() {
    Document document = DocumentHelper.createDocument();
    Element root = document.addElement( "root" );

    Element author2 = root.addElement( "author" )
      .addAttribute( "name", "Toby" )
      .addAttribute( "location", "Germany" )
      .addText( "Tobias Rademacher" );

    Element author1 = root.addElement( "author" )
      .addAttribute( "name", "James" )
      .addAttribute( "location", "UK" )
      .addText( "James Strachan" );

    return document;
  }
}
```

As mentioned earlier dom4j is an interface based API. This means that DocumentFactory and the reader classes in the org.dom4j.io package always use the org.dom4j interfaces rather than any concrete implementation classes. The Collection API and W3C's DOM are other examples of APIs that use this design aproach. This wide spread design is described by [BillVenners].

Chapter 3. Serialization and Output

Once you have parsed or created a document you want to serialize it to disk or into a plain (or encrypted) stream. dom4j provides a set of classes to serialize your dom4j tree in four ways:

- XML
- HTML
- DOM
- SAX Events

Serializing to XML

`org.dom4j.io.XMLWriter` is a easy-to-use and easy-to-understand class used to serialize a dom4jtree to a plain XML. You are able to write the XMLtree to either an `java.io.OutputStream` or a `java.io.Writer`. This can be configured with the overloaded constructor or via the `setOutputStream()` or `setReader()` methods. Let's have a look at a example.

```
import java.io.OutputStream;

import org.dom4j.Document;
import org.dom4j.io.XMLWriter;
import org.dom4j.io.OutputFormat;

public class DeployFileCreator {

 private Document doc;

 public void serializetoXML(OutputStream out, String aEncodingScheme)
   OutputFormat outformat = OutputFormat.createPrettyPrint();
   outformat.setEncoding(aEncodingScheme);
   XMLWriter writer = new XMLWriter(out, outformat);
   writer.write(this.doc);
   writer.flush();
 }

}
```

We use the constructor of `XMLWriter` to pass a to given `OutputStream` along with the required character encoding. It is easier to use a `Writer` rather than an `Out-`

putStream, because the Writer is String based and so has ness character encoding issues. The write() methods of Writer are overloaded so that you can write all of the dom4j objects individually if required.

Customizing the output format

The default output format is to write the XML document as-is. If you want to change the output format then there is a class org.dom4j.io.OutputFormat which allows you to define pretty printing options, to suppress the output of the XML declaration, change the line ending and so on. There is also a helper method OutputFormat.createPrettyPrint() which will create a default pretty printing format that you can further customize if you wish.

```
import java.io.OutputStream;

import org.dom4j.Document;
import org.dom4j.io.XMLWriter;
import org.dom4j.io.OutputFormat;

public class DeployFileCreator {

 private Document doc;

  public void serializetoXML(OutputStream out, String aEncodingScheme
   OutputFormat outformat = OutputFormat.createPrettyPrint();
   outformat.setEncoding(aEncodingScheme);
   XMLWriter writer = new XMLWriter(out, outformat);
   writer.write(this.doc);
   writer.flush();
 }


}
```

An interesting feature of OutputFormat the ability to set the character encoding. It is a good idiom to use this mechansim for setting the encoding as the XMLWriter will be able to use this encoding to create an OutputStream as well as to output the XML declaration.

The close() method closes the underlying Writer.

```
import java.io.OutputStream;
```

```
import org.dom4j.Document;
import org.dom4j.io.XMLWriter;
import org.dom4j.io.OutputFormat;

public class DeployFileCreator {

 private Document doc;
 private OutputFormat outFormat;

 public DeployFileCreator() {
   this.outFormat = OuputFormat.getPrettyPrinting();
 }

 public DeployFileCreator(OutputFormat outFormat) {
   this.outFormat = outFormat;
 }

 public void writeAsXML(OutputStream out) throws Exception {
   XMLWriter writer = new XMLWriter(outFormat, this.outFormat);
   writer.write(this.doc);
 }

 public void writeAsXML(OutputStream out, String encoding) throws Exc
   this.outFormat.setEncoding(encoding);
   this.writeAsXML(out);
 }

}
```

The serialization methods in our little example will now set encoding using `OutputFormater`. The default encoding if none is specifed will be UTF-8. If you need a simple output on screen for debbuing or testing you can omit setting of a `Writer` or an `OutputStream` completely as `XMLWriter` will default to `System.out`.

Printing HTML

`HTMLWriter` takes a dom4j tree and formats it to a stream as HTML. This formatter is similar to `XMLWriter` but outputs the text of CDATA and Entity sections rather than the serialised format as in XML and also supports many HTML element which have no corresponding close tag such as for
 and <P>

```
import java.io.OutputStream;

import org.dom4j.Document;
```

```
import org.dom4j.io.HTMLWriter;
import org.dom4j.io.OutputFormat;

public class DeployFileCreator {

 private Document doc;
 private OutputFormat outFormat;

 public DeployFileCreator() {
   this.outFormat = OuputFormat.getPrettyPrinting();
 }

 public DeployFileCreator(OutputFormat outFormat) {
   this.outFormat = outFormat;
 }

 public void writeAsHTML(OutputStream out) throws Exception {
   HTMLWriter writer = new HTMLWriter(outFormat, this.outFormat);
   writer.write(this.doc);
   writer.flush();
 }

}
```

Building a DOM tree

Sometimes it's necessary to transform your dom4j tree into a DOM tree, because you are currently refactoring your application. dom4j is very convient for integration with older XML API's like DOM or SAX (see
<anchor>Generating SAX Events</anchor>
). Let's move to an example:

```
import org.w3c.dom.Document;

import org.dom4j.Document;
import org.dom4j.io.DOMWriter;

public class DeployFileLoaderSample {

  private org.dom4j.Document doc;

  public org.w3c.dom.Document transformtoDOM() {
    DOMWriter writer = new DOMWriter();
    return writer.write(this.doc);
  }
}
```

Generating SAX Events

If you want to output a document as sax events in order to integrate with some existing SAX code, you can use the `org.dom4j.SAXWriter` clas.

```
import org.xml.ConentHandler;

import org.dom4j.Document;
import org.dom4j.io.SAXWriter;

public class DeployFileLoaderSample {

  private org.dom4j.Document doc;

  public void transformtoSAX(ContentHandler ctxHandler) {
     SAXWriter writer = new SAXWriter();
     writer.setContentHandler(ctxHandler);
     writer.write(doc);
  }
}
```

Using `SAXWriter` is fairly easy as you can see. You can resolve also `org.dom.Element` which means that you are able to process a single element branch or even a single node with SAX.

Chapter 4. Navigation in dom4j

dom4j offers several powerful mechansims for navigating through a document:-

- Using Iterators
- Fast index based navigation
- Using a backed List
- Using XPath
- In-Build GOF Visitor Pattern

Using Iterator

Most Java developers have already used java.util.Iterator or it's ancestor `java.util.Enumeration`. Both classe are fairly involed into the Collection API and used to visit the elements of a collection. The Iterator is appylied usually with a while loop and Iterator methods hasNext() and next() item. Right now Collection API dont support Generic Type (like C++ Templates), but there's already a Early Access Implemention available. Now let's move to an living example of it in dom4j.

```
import java.util.Iterator;

import org.dom4j.Document;
import org.dom4j.Element;

public class DeployFileLoaderSample {

  private org.dom4j.Document doc;
  private org.dom4j.Element root;

  public void iterateRootChildren() {
    root = this.doc.getRootElement();
    Iterator elementIterator = root.elementIterator();
    while(elementIterator.hasNext()){
      Element elmeent = (Element)elementIterator.next();
      System.out.println(element.getName());
    }
  }
}
```

The above example might be a little bit confusing if you are not too familiar with the Collections API. Casting is necessary when you want to acess the object. Java Generics will solve this problem in future.

```
import java.util.Iterator;

import org.dom4j.Document;
import org.dom4j.Element;

public class DeployFileLoaderSample {

  private org.dom4j.Document doc;
  private org.dom4j.Element root;

  public void iterateRootChildren(String aFilterElementName) {
    root = this.doc.getRootElement();
    Iterator elementIterator = root.elementIterator(aFilterElementNar
    while(elementIterator.hasNext()){
      Element elmeent = (Element)elementIterator.next();
      System.out.println(element.getName());
    }
  }
}
```

Now the the method iterates on such Elements that have the *same name* as the parameterized String only. This can be used as a kind of filter applied on top of Collection API's Iterator.

Fast index based Navigation

Sometimes if you need to walk a large tree very quickly, creating an `java.io.Iterator` instance to loop through each `Element`'s children can be expensive in high performance environment. To help this situation, dom4j provides a fast index based looping as follows.

```
  public void treeWalk(Document document) {
    treeWalk( document.getRootElement() );
  }

  public void treeWalk(Element element) {
    for ( int i = 0, size = element.nodeCount(); i < size; i++ ) {
      Node node = element.node(i);
      if ( node instanceof Element ) {
        treeWalk( (Element) node );
```

```
      }
      else {
        // do something....
      }
    }
  }
```

Using a backed List

You can navigate through an `Element`'s children using a backed `List` such that modifications to the `List` are reflected back into the `Element`. It also means that all of the methods on `List` can be used.

```
import java.util.List;

import org.dom4j.Document;
import org.dom4j.Element;

public class DeployFileLoaderSample {

  private org.dom4j.Document doc;

  public void iterateRootChildren() {
    Element root = doc.getRootElement();

    List elements = root.elements;

    // we have access to the size() and other List methods
    if ( elements.size() > 4 ) {
      // now lets remove a range of elements
      elements.subList( 3, 4 ).clear();
    }
  }
}
```

Using XPath

XPath is is one of the most usefull features of dom4j. You can use it to retrieve nodes from any location as well as evaluating complex expressions. A good XPath refercence can be found in Micheal Kay's XSLT book [XSLTReference] along with the [Zvon] Zvon tutorial.

```
import java.util.Iterator;

import org.dom4j.Documet;
```

```
import org.dom4j.DocumentHelper;
import org.dom4j.Element;
import org.dom4j.XPath;

public class DeployFileLoaderSample {

  private org.dom4j.Document doc;
  private org.dom4j.Element root;

  public void browseRootChildren() {
    XPath xpathSelector = DocumentHelper.createXPath("/people/person
    List results = xpathSelector.selectNodes(doc);
    for ( Iterator iter = result.iterator(); iter.hasNext(); ) {
      Element element = (Element) iter.next();
      System.out.println(element.getName();
    }


  }

}
```

As selectNodes returns a List we can apply `Iterator` or any other operation available on `java.util.List`.

Using Visitor Pattern

The visitor pattern has a recrusive behavior and acts like SAX in the way that partical traversal is *not* possible. This means the complete document or the complete element branch will be visited. You should consider wisely when you want to use Visitor pattern, but then it offers a powerful and elegant way of navigation. This document doesn't explain Vistor Pattern in deepth, [GoF98] covers more information.

```
import java.util.Iterator;

import org.dom4j.Visitor;
import org.dom4j.VisitorSupport;
import org.dom4j.Document;
import org.dom4j.Element;

public class VisitorSample {

  public void demo(Document doc) {

    Visitor visitor = new VisitorSupport() {
      public void visit(Element element) {
```

```
        System.out.println(
          "Entity name: " + element.getName()  + "text " + element.ge
        );
      }
    };

    doc.accept( visitor );
  }
}
```

As you can see we used a anonymous inner class to override the `VisitorSupport` callback adapter method visit(Element element) and the accept() method starts the vistor mechansim. Please keep in mind that the *complete* element branch is visited.

Chapter 5. Manipulating dom4j

Accessing XML content statically alone would not very special. Thus dom4j offers several methods for manipulation a documents content.

What `org.dom4j.Document` provides

A `org.dom4j.Document` allows you to configure and retreive the root element. You are also able to set the DOCTYPE or a SAX based `EntityResolver`. An empty `Document` should be created via `org.dom4j.DocumentFactory`.

Working with `org.dom4j.Element`

`org.dom4j.Element` is a powerfull interface providing lots of methods for manipulation an Element.

```
public void changeElementName(String aName) {
  this.element.setName(aName);
}

public void changeElementText(String aText) {
  this.element.setText(aText);
}
```

Qualified Names

An XML Element should have a qualified name. A qualified name consits normally of a Namespace and a local name. It's recommend to use `org.dom4j.DocumentFactory` to create Qualifed Names that are provided by `org.dom4j.QName` instances.

```
import org.dom4j.Element;
import org.dom4j.Document;
import org.dom4j.DocumentFactory;
import org.dom4j.QName;

public class DeployFileCreator {
```

```
  protected Document deployDoc;
  protected Element root;

  public void DeployFileCreator()
  {
    QName rootName = DocumentFactory.getInstance().createQName("pref
    this.root = DocumentFactory.getInstance().createElement(rootName
    this.deployDoc = DocumentFactory.getInstance().createDocument(th
  }
}
```

Inserting elements

Somethimes it's necessary to insert an element somewhere in a existing XML Tree. As dom4j is based on Collection API this causes no problems. The following expample shows how it could be done.

```
public void insertElementAt(Element newElement, int index) {
  Element parent = this.element.getParent();
  List list = parent.content();
  list.add(index, newElement);
}

public void testInsertElementAt() {

//insert an clone of current element after the current element
  Element newElement = this.element.clone();
  this.insertElementAt(newElement, this.root.indexOf(this.element

// insert an clone of current element before the current element
  this.insertElementAt(newElement, this.root.indexOf(this.element
}
```

Studying the Collection API should lead to more solutions for similar problem and you will notify that dom4j fits well in the Collection Framework and both complement each other in order to processing xml document in a comfortable way.

Cloning - How many sheeps do you need?

Elements can be cloned as well. Usually cloning is supported in Java with clone()

method that is derived from `Object`, but a cloneable Object have to implement interface `Clonable`. Java support shallow copying by simply returning *this* for standard. dom4j supporting deep cloning because shallow copies would not make sence in context of an XML object model. This means that cloning can take a while because the complete tree branch or event the document will be cloned. Now we have a short look *how* dom4j cloning mechanism is used.

```
import org.dom4j.Document;
import org.dom4j.Element;

public class DeployFileCreator {


 private Element cloneElement(String name) {
  return this.root.element(name).clone();
 }

 private Element cloneDetachElement(String name) {
   return this.root.createCopy(name);
 }

 public class TestElement extends junit.framework.TestCase {

   public void testCloning() throws junit.framwork.AssertionFailedE
     assert("Test cloning with clone() failed!", this.creator.clone
     assert("Test cloning with createCopy() failed!", this.creator
   }
 }
}
```

The difference between *createCopy(...)* and *clone()* is that first is a polymorphic method that created a decoupled deep copy whereas *clone()* returns a returns a deep copy of the current document or element itself.

Consider use of Cloning

Cloning might be usefull when you want to build a element pool. Such a pool should be desinged carefully keeping `OutOfMemoryException` in mind. You could alternativly consider to use Reference API [Pawlan98] or Dave Millers approach [JavaWorldTip76].

Chapter 6. Using dom4j with XSLT

With eXtensible Stylesheet Language XML got's a powerfull method of transforming itself into other formats. Developing Exportfilter's for dataformats are normally a hard job and so for XML XSL simpliefs that work. The aronym XSLT means the process of transformation, that is usally done by an XSL compliant Processor. XSL covers following subjects:

- XSL Style Sheet
- XSL Processor for XSLT
- FOP Processor for FOP
- An XML source

Since JaXP 1.1 TraX is the common API for proceeding a XSL Stylesheet inside of Java. You start with a `TransformerFactory` and dealing with `Result` and `Source`. A `Source` contains the source xml file that should be transformed. `Result`'s contains the the result of transformation. dom4j offers `org.dom4j.io.DocumentResult` and `org.dom4j.io.DocumenSource` for compatiblity to TrAX. Whereas `org.dom4j.io.DocumentResult` contains a `org.dom4j.Document` as result tree, `DocumentSource` takes dom4j `Document`s and pepare them for transformation. Both classes are build on top of TraX own SAX classes. This is much more perfomant as a DOM adaptation. The following example explains the use of XSLT with TraX and dom4j.

```
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamSource;

import org.dom4j.Document;
import org.dom4j.io.DocumentResult;
import org.dom4j.io.DocumentSource;

public class DocumentStyler
{
    private Transformer transformer;

    public DocumentStyler(Source aStyleSheet) throws Exception {
        // create the transformer
        TransformerFactory factory = TransformerFactory.newInstance(
        transformer = factory.newTransformer( aStyleSheet );
    }
```

```
    public Document transform(Document aDocument) throws Exception {

        // perform the transformation
        DocumentSource source = new DocumentSource( aDocument );
        DocumentResult result = new DocumentResult();
        transformer.transform(source, result);

        // return the resulting document
        return result.getDocument();
    }
}
```

Imagine that you use XSLT to process a XML Schema in order to generate a empty template xml file accoring the schema contraints. The above sample should how easy the Java code is when you use dom4j and it's TraX support. If you use TemplateGenerator a lot you should consider the application of singleton pattern, but for this example I avoided this for simplicity. More information about TraX is provided here [http://www.java.sun.com/xml].

Chapter 7. Schemata-Support

The first way to describe and constrain the form and data of a XML document is as old as XML itself. Document Type Definitions are used since the XML Specicifiation has been published. At lot of applications used this DTDs to describe and validate her documents. Unfortunatly the DTD Syntax was not that powerfull as needed. Written in SGML, DTDs are also not so easy to handle as XML is.

During the time of DTDs a couple of people invents several other possible ways that could be used to describe a document and force its content in the desired form. Latly the W3C published XML Schema Specification with a couple of massive inprovements. XML Schemas are no described by XML and the way to describe a Schema is done with DTD once and not longer by every XML user. A growing group of people using XML Schema now. But XML Schema isn't perfect. So a few people swear by Relax or Relax NG. The reader of this document is able to choose one of the following technologies:

- Relax NG (Regular Language description for XML Next Generation)[RelaxNG]
- Relax (Regular Language description for XML)[Relax]
- TREX[TREX]
- XML DTDs[DTD]
- XML Schema[XSD]

Using XML Schema Data Types in dom4j

dom4j supports currently XML Schema Data Types[DataTypes] only. The dom4j implementation is based on top of MSV. Earlier dom4j releases are built on top of Suns Tranquilo (xsdlib.jar) but we use MSV now, because its uses offers the same Tranquilo plus exiting additional features we discuss later.

```
import java.util.List;

import org.dom4j.Document;
import org.dom4j.DocumentHelper;
import org.dom4j.XPath;
import org.dom4j.io.SAXReader;
import org.dom4j.dataType.DataTypeElement;

public class SchemaTypeDemo {
```

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

  SAXReader reader = new SAXReader();
  reader.setDocumentFactory( DatatypeDocumentFactory.getInstance() )
  Document schema =  return reader.read(xmlFile)
  XPath xpathSelector = DocumentHelper.createXPath("xsd:schema/xsd:co
  List xsdElements = xpathSelector.selectNodes(schema);

  for (int i=0; i < xsdElements.size(); i++) {
    DataElement tempXsdElement = (DatatypeElement)xsdElements.get(i)

    if (tempXsdElement.getData() instanceof Integer) {
       tempXsdElement.setData(new Integer(23));
     }
  }
}
```

Alpha status

Note that the Data Type support is still alpha. If you find any bug, please report it to the mailing listy. This helps us to make the Data Type support more error-prone and trustworthy.

Validation

dom4j currently comes not with its one validation engine. You are forced to use a different engine instead. We recommend the use of Xerces 1.4.x or later in the past, but now you are able to use Suns Multi Schema Validator as well. Xerces is able to validate against DTDs and XML Schema, but not against TREX or Relax. You would belive it, but the Mulit Schema Validator Libery supports all earlier mentioned types for validation.

Consider use of Validation

Using Valdiation consumes valueable resources. Use it wisely.

Using Apaches Xerces 1.4.x and dom4j for validation

Using Xerecs 1.4.x for Schema and Validation is very easy. You have to download Xerces at Apaches XML web sites. The past has shown that not always the newest version is best. You can view there mailing lists in order to find out which version is buggy and which works well. For Schema support at least Xerecs 1.4.0 is necessary. If you work according the following rules valdation should be no problem.

- Turn on validation mode - which is false for default - using a SAXReader instance
- Set the following Xerces property http://apache.org/xml/properties/schema/external-noNamespaceSchemaLocation using the schema URI.
- Create a SAX XMLErrorHandler and install it to your SAXReader instance.
- Parse and validate the Document.
- Output Validation/Parse Errors errors.

```
import org.dom4j.Document;
import org.dom4j.Element;
import org.dom4j.io.OutputFormat;
import org.dom4j.io.SAXReader;
import org.dom4j.io.XMLWriter;
import org.dom4j.util.XMLErrorHandler;


import org.xml.sax.ErrorHandler;
import org.xml.sax.SAXParseException

public class SimpleValidationDemo {

public static void main(String[] args) {
  SAXReader reader = new SAXReader();

  reader.setValidation(true);

  // specify the schema to use
  reader.setProperty(
   "http://apache.org/xml/properties/schema/external-noNamespaceSchem
   "prices.xsd"
  );

  // add an error handler which turns any errors into XML
   XMLErrorHandler errorHandler = new XMLErrorHandler();
   reader.setErrorHandler( errorHandler );

  // now lets parse the document
  Document document = reader.read(args[0]);

 // now lets output the errors as XML
  XMLWriter writer = new XMLWriter( OutputFormat.createPrettyPrint()
  writer.write( errorHandler.getErrors() );
```

```
}
```

Xerces and Crimson

Both, Xerecs and Crimson, are JaXPable Parsers. You should be carefully in using Crimson and Xerces in same classpath. Xerces will work only correct when it is the mentioned *before* Crimson in classpath. At this time I recommend that you should either Xereces *or* Crimson.

A perfect team - Multi Schema Validator[MSV] and dom4j

Kohsuke Kawaguchi a deleloper from Sun created a extremly usefull tool for validtion of XML documents. The Multi Schema Validator (MSV) supports following kinds of Schmemata:

- Relax NG
- Relax
- TREX
- XML DTDs
- XML Schema

You are able to use the MSV and dom4j in order to validate your Documents. The following examples shows you how to use the MSV and with dom4j.

```
import com.sun.msv.grammar.Grammar;
import com.sun.msv.reader.util.GrammarLoader;
import com.sun.msv.reader.util.IgnoreController;
import com.sun.msv.verifier.DocumentDeclaration;
import com.sun.msv.verifier.ValidityViolation;
import com.sun.msv.verifier.Verifier;
import com.sun.msv.verifier.VerificationErrorHandler;

import javax.xml.parsers.SAXParserFactory;

import org.dom4j.Document;
import org.dom4j.DocumentException;
import org.dom4j.io.SAXReader;
import org.dom4j.io.SAXWriter;
```

```
import org.xml.sax.ContentHandler;
import org.xml.sax.ErrorHandler;
import org.xml.sax.Locator;
import org.xml.sax.SAXParseException;

import java.net.URL;
import java.io.File;

public class Schema {

    public static void main(String argv[]) {
        try {
            String filename = argv[0];
            String schema = argv[1];

            URL fileURL = new File(filename).toURL();
            URL schemaURL = new File(schema).toURL();

            SAXReader reader = new SAXReader();
            Document doc = reader.read(fileURL);
            validate(doc, schemaURL.toExternalForm());
        } catch (Exception e) {
            e.printStackTrace(System.err);
        }
    }

    static public void validate(Document doc, String schema)
        throws Exception {

        // Turn on Namespace handling in theJAXP SAXParserFactory
        SAXParserFactory saxFactory = SAXParserFactory.newInstance();
        saxFactory.setNamespaceAware(true);

        // create MSVs DocumentDeclaration by overriding
        // a IgnoreController in an anonyous inner class
        DocumentDeclaration docDeclaration =
            GrammarLoader.loadVGM(schema, new IgnoreController() {

                    public void error(Locator[] locations,
                                      String message,
                                      Exception exception) {
                        System.out.println("ERROR: " + message);
            }

            public void error(Locator[] locations, String message) {
                System.out.println("WARNING: " + message);
            }
        }, saxFactory);
```

```
        // create a new Verifier that reports vlidation errors
        // using an anonymous inner class
        Verifier verifier =
            new Verifier(docDeclaration, new VerificationErrorHandler

                public void onError(ValidityViolation e) {
                    System.out.println("Document invalid! Error:
                }

                public void onWarning(ValidityViolation e) {
                    System.out.println("Document invalid! Warning
                }
            });

        SAXWriter writer = new SAXWriter((ContentHandler) verifier);
        writer.setErrorHandler(new ErrorHandler() {

                public void error(SAXParseException e) {
                    System.out.println("ERROR:" + e);
                }

                public void fatalError(SAXParseException e) {
                    System.out.println("Fatal:" + e);
                }

                public void warning(SAXParseException e) {
                    System.out.println("Warning:" + e);
                }
        });

        // validate now!
        writer.write(doc);
        if (verifier.isValid())
            System.err.println("The document was valid");
        else
            System.err.println("The document was not valid");
    }
}
```

At the first look the use of MSV looks not trival. The Xerces validation is easier to use in code, but not so powerfull. Currently its not clear if XML Schema will be the next standard for validation. Relax NG gots a even more growing lobby. If you want to build a open application that is not fixed to a specific XML parser and specific XML Schematas you should use this powerfull tool.

Further Reading

Books

[XSLTReference] Michael Kay. Copyright © 2001 Worx Press, Inc.. 1-861-005067. Worx Press. *XSLT Programmer's Reference 2'nd Edition*. *Programmer To Programmer*. Worx Press.

[GoF95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Copyright © 1995 Addison Wesley Pub, Co.. 0-201-633-612. Addison-Wesley. *Design Patterns: Elements of Reusable Object-Orientated Software* .