SEMINAR:
XML und Datenbanken

XML-Verarbeitungsmodelle und Language Bindings

Christian Müller

WS 2002/2003

Inhaltsverzeichnis

1		**Einleitung**	**3**
2		**Simple API for XML (SAX)**	**3**
	2.1	Allgemeines zu SAX	3
	2.2	Beispiel	4
	2.3	Bewertung	8
3		**Document Object Model (DOM)**	8
	3.1	Allgemeines zu DOM	8
	3.2	Beispiel	9
	3.3	Bewertung	12
4		**JDOM**	**12**
	4.1	Allgemeines zu JDOM	12
	4.2	Unterschiede zu DOM	15
5		**Java API for XML Processing (JAXP)**	**15**
	5.1	Allgemeines zu JAXP	15
	5.2	Beispiele zur Verwendung von SAX und DOM	16
6		**XL**	**17**
	6.1	Allgemeines zu XL	17
	6.2	Beispiel	18
7		**Zusammenfassung**	**19**
8		**Literatur**	**20**

1 Einleitung

XML gewinnt in der Anwendungsentwicklung zunehmend mehr an Bedeutung, da es zwei wichtige Eigenschaften besitzt: Portierbarkeit und Flexibilität. Da XML-Dokumente als Textdateien verwaltet werden, können diese zwischen den verschiedensten Plattformen ausgetauscht werden. Die zweite Eigenschaft ist die Flexibilität von XML, da nur wenige Vorgaben über die einzelnen Elemente und Inhalte eines Dokuments gemacht werden. Diese Eigenschaft ist besonders wichtig für Unternehmensanwendungen, da die Interoperabilität zwischen unterschiedlichen Komponenten verbessert wird.

Um die XML-Daten nun in den Anwendungen nutzen zu können, benötigt man eine Möglichkeit, die Daten zu verarbeiten. Dazu werden in dieser Seminararbeit die Simple API for XML (SAX), das Document Object Model (DOM) und JDOM vorgestellt. SAX erzeugt mit Hilfe eines Parsers für jedes Element des Dokuments ein Ereignis. Beim Eintreten eines Ereignisses kann der Programmierer anwendungsspezifischen Code ausführen. In dem sprachunabhängigen DOM werden die XML-Dokumente in einer Baumstruktur dargestellt, wir werden hier hauptsächlich die Sprachbindungen für Java besprechen. JDOM, welches für und in Java entwickelt wurde, arbeitet ähnlich wie DOM mit einer Baumstruktur. Eine Abstraktionsschicht über SAX und DOM liegt die Java API for XML Processing (JAXP), eine von Sun entwickelte Java-API. JAXP stellt keine neuen Mittel zur Verfügung, sondern erleichtert die Lösung einiger Aufgaben in SAX und DOM auf herstellerunabhängige Weise.

Abschließend behandeln wir noch XL, eine XML-Programmiersprache, welche speziell für die Implementierung von Web Services entwickelt wurde.

2 Simple API for XML (SAX)

2.1 Allgemeines zu SAX

In diesem Kapitel wird SAX in der Version 2.0 beschrieben. SAX ist eine Java-API zum ereignisbasierten Parsen von XML-Dokumenten. Beim Verarbeiten von XML mit Hilfe von SAX treten Ereignisse (Events) auf, wodurch vom SAX Prozessor Methoden des Interface *org.xml.sax.ContentHandler* aufgerufen werden. Diese Methoden werden auch Callbacks genannt.

Zu Beginn eines XML-Dokuments wird die Methode *setDocumentLocator,* gefolgt von der Methode *startDocument* aufgerufen. Am Ende des Dokuments wird *endDocument* ausgeführt. Zu den Beginn- und Ende-Tags eines Elements werden die Methoden *startElement* bzw. *endElement*, beim Parsen von Textdaten wird die Methode *characters* und beim Auftreten von Leerzeichen wird die Methode *ignorableWhitespace* aufgerufen. Die Callbacks *startPrefixMapping* und *endPrefixMapping* werden vor bzw. nach der Deklaration eines neuen Namensraums ausgelöst. Beim Auftreten einer ProcessingInstruction wird *processingInstruction* und beim Auftreten von einer Entity-Referenz wird die Methode *skippedEntity* aufgerufen.

Betrachtet man das in Abbildung 1 dargestellte XML-Fragment, so wird zu Beginn des Dokuments *startDocument* aufgerufen, nach *<seminar>, <vortrag>, <vortragender> usw.* wird *startElement* aufgerufen. Beim Auftreten von Text (z.B. Weber) wird die Methode *characters* aufgerufen. Der Programmierer kann das Interface *ContentHandler* implementieren, in den entsprechenden Methoden seinen An-

wendungscode einfügen und mit den XML-Daten die gewünschten Aktionen durchführen.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE sem SYSTEM "beispiel.dtd">
<seminar>
  <vortrag thema="XML-Grundlagen">
    <vortragender>
      <name>Weber</name>
      <vorname>Christian</vorname>
    </vortragender>
  </vortrag>
  <vortrag thema="XML-Verarbeitungsmodelle">
    <vortragender>
      <name>Müller</name>
      <vorname>Christian</vorname>
    </vortragender>
  </vortrag>
  <vortrag thema="Repräsentation und Struktur">
    <vortragender>
      <name>Tuchbreiter</name>
      <vorname>Jochen</vorname>
    </vortragender>
  </vortrag>
...
</seminar>
```

Abbildung 1: *beispiel.xml*

Am Beispiel des XML-Dokuments beispiel.xml in Abbildung 1 werden wir nun im Folgenden die Grundfunktionalität von SAX an einem Beispiel kennen lernen. Auch in den späteren Kapiteln werden wir dieses Dokument für einige Beispiele wieder verwenden.

2.2 Beispiel

Zu Beginn unseres Beispiels benötigen wir einen Parser, der die Verarbeitung der XML-Dokumente durchführt. Hier werden wir Apache Xerces [2] in der Java-Version 2.2.1 verwenden, der zu SAX 2 kompatibel ist. Es gibt noch viele andere kompatible Parser, so zum Beispiel:

- IBM XML4J [4]
- Oracle XML Parser [5]
- Crimson von Sun Microsystems [6]

Der Download von Xerces enthält über den Parser hinweg auch noch die SAX Klassen und Interfaces, womit alles, was wir für dieses Beispiel benötigen, vorhanden ist.

Wir werden nun sehen, wie man ein XML-Dokument mit Hilfe von SAX und dem Parser Xerces einlesen und verarbeiten kann. Dies erfolgt bei anderen Parsern

analog. Wir werden einige Ereignis-Callbacks mit Leben füllen, um die eingelesenen Daten mittels *System.out* wieder auszugeben. Unser Beispiel wird aus drei Klassen bestehen, welche nun erläutert werden.

Beginnen wir mit der Hauptklasse *SaxBeispiel.java*, in der die Methode *void startParsing(String xmlUri)* definiert ist. In dieser Methode müssen wir zuerst eine Instanz eines XMLReaders erzeugen. Dies ist der einzige Punkt, an dem wir auf den Parser zugreifen.

```
XMLReader reader = XMLReaderFactory.
         createXMLReader("org.apache.xerces.parsers.SAXParser");
reader.setContentHandler(new MyContentHandler());
reader.setErrorHandler(new MyErrorHandler());
reader.setFeature("http://xml.org/features/validation",true);
InputSource input = new InputSource(xmlUri);
reader.parse(input);
```

Abbildung 2: Methode *startParsing*

Der Aufruf der Methode *createXMLReader* erzeugt aus der Klasse *org.apache.xerces.parsers.SAXParser* einen XMLReader. Bei der Nutzung eines anderen Parsers muss die Klassenangabe durch die abweichende Parserklasse ersetzt werden. In dieser Methode registrieren wir noch einen ContentHandler mit der Klasse *MyContentHandler* und einen ErrorHandler mit der Klasse *MyErrorHandler* bei dem XMLReader. Diese beiden Handler-Klassen werden wir uns gleich noch anschauen. Durch *setFeature* können Eigenschaften des Parsers eingestellt werden. Die spezifizierte URL *"http://xml.org/features/validation"* ist eine Konstante, die dafür steht, dass die Validierung anhand einer DTD-Datei eingeschaltet wird. Dabei wird nicht versucht, diese Internetadresse aufzurufen. Im Beispiel wird gegen die Datei *beispiel.dtd* validiert, auf welche in der XML-Datei aus Abbildung 1 verwiesen wird. Dann wird eine *org.xml.sax.InputSource* erzeugt, welche die Adresse unseres XML-Dokuments kapselt und dem Parser Informationen über die Eingabequelle, wie beispielsweise eine System-Id oder einen Byte-Stream des Dokuments zur Verfügung stellt. Danach starten wir durch den Aufruf von *parse* das Parsen des Dokuments. Bevor dies funktioniert, müssen wir noch die Klassen *MyContentHandler* und *MyErrorHandler* implementieren.

Beginnen wir mit *MyContentHandler,* welche das SAX-Interface *ContentHandler* implementiert, womit natürlich alle dort definierten Methoden implementiert werden müssen. Zuerst speichern wir das *org.xml.sax.Locator* Objekt in einer Variablen, was zu Beginn des Dokuments durchgeführt wird.

```
class MyContentHandler implements ContentHandler {
  Locator locator;
  public void setDocumentLocator(Locator locator) {
    this.locator = locator; }
```

Abbildung 3: Methode *setDocumentLocator*

Dieser Locator dient dazu, den anderen Methoden der Handler-Klassen Informationen über die Positionen (mit Methoden wie getLineNumber() oder getColumnNumber()) an denen Ereignisse auftreten, zur Verfügung zu stellen. Weiterhin sind

die Callback-Methoden zu implementieren, die zu Beginn und Ende des Dokuments aufgerufen werden.

```
public void startDocument () throws SAXException {
  System.out.println(“Start”); }
```

Abbildung 4: Methode *startDocument*

Analog wird die Methode *endDocument()* implementiert, nur werden wir dort den Text “Ende” ausgeben. Zwei wichtige Callbacks sind die nun folgenden Methoden.

```
public void startElement(String namespaceURI, String localName,
            String qName, Attributes atts) throws SAXException {
  System.out.println(“Elementname: “+localName);
  for (int i=0;i<atts.getLength();i++) {
    System.out.println(“Attributname: “ + atts.getLocalName(i) +
        “ Wert: “ + atts.getValue(i)); } }

public void endElement(String namespaceURI, String localName,
            String qName) throws SAXException {
  System.out.println("endElement: " + localName); }
```

Abbildung 5: Methoden *startElement* und *endElement*

Die Methoden *startElement* bzw. *endElement* werden jeweils beim Beginn bzw. Ende eines Elements aufgerufen. Mit dem XML-Dokument aus Abbildung 1 wäre dies beim Auftreten von *<seminar>*, *<vortrag>*, usw. und bei den entsprechenden Ende-Tags der Fall. Bei den Aufrufen enthält der *localName* den Namen des Elements ohne Präfix, wogegen der *qName* den Namen mit Präfix darstellt. Der *qName* entspricht also dem Elementnamen, welcher in der XML-Datei steht. Das Präfix ist gleich der Namensraumadresse und wird in *namespaceURI* übergeben. Die Variable *atts* enthält die Attribute des Elements und verwaltet sie in einer *Vector*-ähnlichen Struktur. In unserem Beispiel haben die Elemente keine Präfixe, somit sind *localName* und *qName* gleich. Einen Namensraum wurde in unserem XML-Dokument auch nicht zugeteilt, somit ist der String leer. Wir geben hier den Namen des Elements aus und iterieren anschließend über alle Attribute des Elements, um auch diese auszugeben. Beim Aufruf von *endElement* geben wir ebenfalls den Elementnamen aus.

Durch die Methode *characters* werden die Textdaten eines Elements an die Anwendung geliefert. Wir geben wiederum nur die Textdaten aus, was wie folgt aussieht:

```
public void characters(char[] ch, int start, int length)
      throws SAXException {
  String s = new String(ch, start, length);
  System.out.println("Characters: " + s); }
```

Abbildung 6: Methode *characters*

Die weiteren Callback-Methoden werden wir nun in einer verkürzten Version besprechen, da in diesem Beispiel die Implementierungen meist keinen Code enthalten. So gibt es die Methode *ignorableWhitespace,* welche beim Auftreten von Leerzeichen aufgerufen wird. Die Methode *skippedEntity* lassen wir leer, da in unserem Dokument keine Entity-Referenzen existieren. Die Methode ist vorhanden, um von nicht-validierenden Parsern übergangene Referenzen zu erhalten. Die weiteren Methoden, deren Implementierung wir leer lassen, sind *processingInstruction, startPrefixMapping* und *endPrefixMapping.* Die erste Methode wird beim Auftreten einer ProcessingInstruction aufgerufen, um diese verarbeiten zu können. Die *startPrefixMapping*-Methode wird direkt vor dem Auftreten eines Elements aufgerufen, welches einen neuen Namensraum deklariert. Nach dem deklarierenden Element wird die Methode *endPrefixMapping* aufgerufen. Diese Aufrufreihenfolge stellt eine Besonderheit in SAX dar, da es eine Abweichung von der eigentlich sequentiellen Abarbeitung darstellt.

Die letzte Klasse, die wir noch benötigen, ist die Klasse *MyErrorHandler,* welche das Interface *org.xml.sax.ErrorHandler* implementiert. Dieses Interface hat die drei Methoden *warning, error* und *fatalError,* die implementiert werden müssen. Die Methoden werden aufgerufen, wenn eine Warnung, ein kritischer oder nicht-kritischer Fehler auftritt. Ein kritischer Fehler ist beispielsweise ein fehlender Ende-Tag zu einem Element. Da in unserem Beispiel alle drei Methoden sehr ähnlich aussehen, werden wir nur die Methode *fatalError* genauer betrachten.

```
public void fatalError(SAXParseException exception)
            throws SAXException {
  System.out.println("**Parsing Fatal Error**\n" + " Zeile: " +
         exception.getLineNumber() + "\n" + " URI: " +
         exception.getSystemId() + "\n" + " Mitteilung: " +
         exception.getMessage());
  throw new SAXException("Fatal Error encountered"); }
```

Abbildung 7: Methode *fatalError*

Die Methode *fatalError* erhält eine *SAXParseException*, welche Informationen über den aufgetretenen Fehler bereithält, wie die Zeile, den Dateinamen (SystemId) und eine Nachricht über den Fehler. Diese Informationen geben wir aus. Dann wirft die Methode eine SAXException, wodurch der Fehler weiter nach oben gereicht wird und bearbeitet werden kann.

Zum Abschluss unseres Beispiels erzeugen wir eine Instanz von *SaxBeispiel* und rufen die Methode *startParsing("beispiel.xml")* auf. Starten wir nun unser Programm, so erhalten wir folgende Ausgabe:

Start

Elementname: seminar

Elementname: vortrag

Attributname: thema Wert: XML-Grundlagen

Elementname: vortragender

Elementname: name

Characters: Weber ...

Diese Ausgabe lässt sich im gleichen Stil fortsetzten und spiegelt das eingelesene Dokument wider.

2.3 Bewertung

SAX ist nicht zur Änderung und Ausgabe von XML geeignet. Dies ist nicht vorgesehen, und es gibt auch keine Klassen in SAX, welche das Schreiben von XML-Daten unterstützen. Sicherlich kann man dies trotzdem mit Hilfe von SAX durchführen, aber dazu sind andere XML-APIs besser geeignet. Ein wichtiger Vorteil ist die sequentielle Datenverarbeitung von SAX. So werden die Daten direkt beim Einlesen verarbeitet, wodurch Systemressourcen, insbesondere Speicher, gespart werden können. Andere APIs, wie DOM und JDOM, haben in diesem Bereich deutliche Nachteile, wie wir in Kapitel 3 sehen werden. Die sequentielle Verarbeitung hat aber auch den Nachteil, dass man beim Verarbeiten eines Knotens nicht mehr auf den letzten oder vorletzten Knoten zugreifen kann, ohne ihn selbst zwischenzuspeichern.

3 Document Object Model (DOM)

3.1 Allgemeines zu DOM

Das Document Object Model [3] ist im Gegensatz zu SAX nicht an eine Programmiersprache gebunden. DOM ist ein sprachunabhängiges Modell, um den Inhalt und die Struktur von Dokumenten, insbesondere XML-Dokumenten, darzustellen. Wir werden hier DOM Level 2 besprechen, wobei man Level hier auch durch Version ersetzen könnte. Um DOM auch in verschiedenen Programmiersprachen nutzen zu können, gibt es verschiedene Sprachbindungen zu DOM, beispielsweise für CORBA, JavaScript und auch Java. Diese Sprachbindungen dienen als API für die entsprechende Programmiersprache, um die Dokumente gemäß den DOM-Spezifikationen verarbeiten zu können.

Den Kern von DOM bildet ein Baummodell, welches die typische Struktur eines XML-Dokuments darstellt. Dieses Baumformat baut vollständig auf dem DOM-Interface *org.w3c.dom.Node* auf. Andere Interfaces wie *Document, Element, Attr* oder *Text* sind von diesem Node-Interface abgeleitet. Das Beispieldokument in Abbildung 1 sieht in der DOM-Baumstruktur folgendermaßen aus:

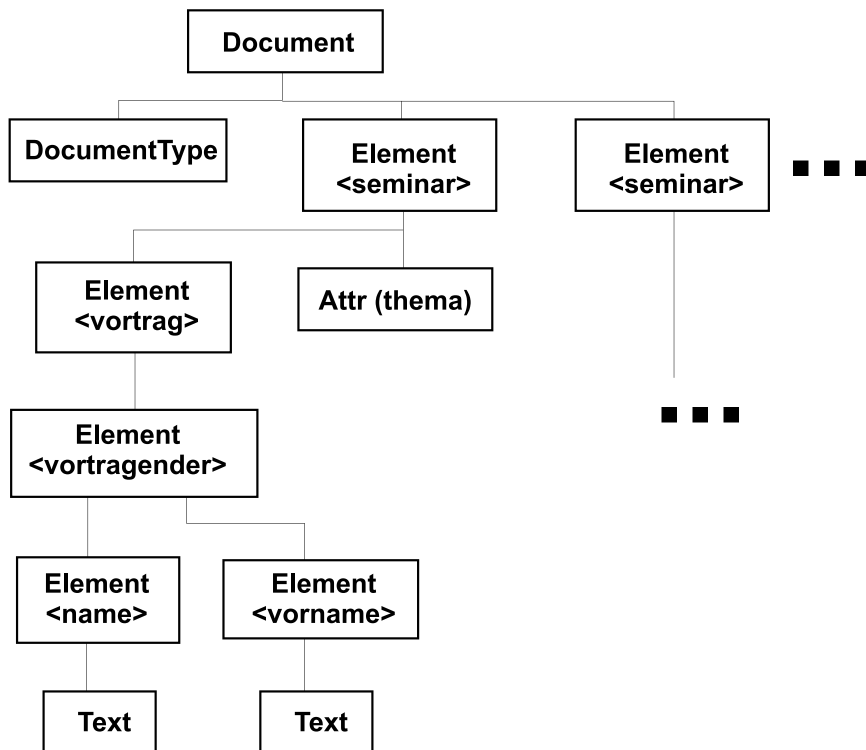

Abbildung 8: DOM-Baum zu Abbildung 1

Im Gegensatz zu SAX, welches Ereignisse während des Parsens meldet, ist die DOM-Baumstruktur erst nach dem kompletten Parsen des Dokuments verfügbar. Dies hat Vor- und Nachteile, die wir später noch genauer besprechen werden. Man kann nach dem Parsen mittels *getDocument()* den Wurzelknoten des Dokuments ermitteln und dann selbstständig den Baum traversieren. Im Folgenden werden wir uns nun die Java-Sprachbindungen anhand eines Beispiels etwas genauer anschauen.

3.2 Beispiel

Um DOM nutzen zu können, benötigen wir einen Parser. Wir greifen wieder auf Apache Xerces zurück. Der Download beinhaltet die benötigten Klassen und Interfaces von DOM. In diesem Beispiel werden wir, ähnlich wie im vorherigen Beispiel, die Struktur und Daten des Beispieldokuments mittels *System.out* ausgeben. Hier werden allerdings keine Callbacks implementiert, sondern wir erhalten nach dem Parsen des kompletten Dokuments eine Instanz von *org.w3c.dom.Document,* die das Dokument repräsentiert. Mit dieser Instanz werden wir dann mittels Rekursion die einzelnen Knoten abarbeiten.

Zu Beginn müssen einige Klassen und Interfaces importiert werden. Eine davon ist die Klasse *org.apache.xerces.parsers.DOMParser,* womit klar ist, dass der Code von dem verwendeten Parser abhängig ist. Wie man DOM parserunabhängig verwenden kann, werden wir später im Kapitel über JAXP noch sehen.

In unserer *main*-Methode erzeugen wir zuerst einen Parser, parsen das Document und lassen uns dann mit Hilfe der Methode *getDocument()* der Klasse *DOMParser* ein *Document*-Objekt als Einstiegspunkt geben.

```
DOMParser parser = new DOMParser();
parser.parse("beispiel.xml");
Document doc = parser.getDocument();
```

Abbildung 9: Parsen von *beispiel.xml*

Mit diesem *Document*-Objekt als Startpunkt rufen wir nun die einzige weitere Methode auf, die wir noch benötigen. Der Methode *printNode(Node node)* wird ein Knoten übergeben. Wir übergeben zwar ein *Document*-Objekt, aber *org.w3c.dom.Document* ist die Subklasse von *org.w3c.dom.Node* womit es auch als *Node* behandelt werden kann. Zu Beginn der *printNode*-Methode ermitteln wir den Typ des übergebenen Knotens und beginnen eine switch-Anweisung.

```
public void printNode(Node node) {
  switch (node.getNodeType()) {...}
```

Abbildung 10: Methode *printNode*

In dieser switch-Anweisung werden wir nun die einzelnen Knotentypen durcharbeiten. Beginnen werden wir nun mit einem Knoten des Typs *Document.*

```
case Node.DOCUMENT_NODE:
  NodeList nodes = node.getChildNodes();
  if (nodes != null) {
    for (int i=0; i<nodes.getLength(); i++) {
     printNode(nodes.item(i)); } }
  break;
```

Abbildung 11: Dokumentknoten

Nachdem wir einen Dokument-Knoten erkannt haben, lassen wir uns alle Kindknoten des aktuellen Knotens zurückgeben und speichern diese in einer *org.w3c.dom.NodeList.* Diese Liste wird dann in einer for-Schleife durchiteriert und für jeden Knoten die Methode *printNode* aufgerufen, womit wir die Rekursion starten. Als nächsten Knoten in der switch-Anweisung werden wir den Elementknoten behandeln.

```
case Node.ELEMENT_NODE:
  System.out.println("Element: " + node.getNodeName());
  NamedNodeMap attributes = node.getAttributes();
  for (int i=0; i<attributes.getLength(); i++) {
    Node current = attributes.item(i);
    System.out.println("Attributname: " + current.getNodeName()
                  + " Wert: " + current.getNodeValue()); }
  NodeList children = node.getChildNodes();
  if (children != null) {
    for (int i=0; i<children.getLength(); i++) {
      printNode(children.item(i)); } }
  System.out.println("endElement: " + node.getNodeName());
  break;
```

Abbildung 12: Elementknoten

Zuerst geben wir nach erkanntem Elementknoten aus, dass es sich um ein Element handelt. Danach geben wir den Elementknoten aus, lassen uns von dem Knoten mittels *getAttributes()* die Attribute des Elements ermitteln und speichern sie in einer *org.w3c.dom.NamedNodeMap*. Diese Attribute werden dann in einer for-Schleife durchiteriert und die Namen, sowie die Werte der Attribute ausgegeben. Damit sind die Attribute abgearbeitet, und wir kommen zu den Kindknoten unseres Elementknotens. Diese werden wie beim Dokumentknoten auch in einer *NodeList* gespeichert. Rekursiv wird für jeden Kindknoten die Methode *printNode* aufgerufen. Nachdem diese Knoten abgearbeitet sind, kommt im XML-Dokument das schließende Element-Tag, und dementsprechend geben wir hier "endElement" gefolgt von dem Elementnamen aus. Nun haben wir alle Knoten mit Kindknoten abgearbeitet, und wir kommen zum Textknoten.

```
case Node.TEXT_NODE:
  System.out.println("Text: " + node.getNodeValue());
  break;
```

Abbildung 13: Textknoten

Für diesen Knoten geben wir den im Knoten gespeicherten Text aus. In diesem Beispiel werden wir keine weiteren Knotentypen betrachten. In Abbildung 14 folgt eine Tabelle über die in DOM vorhandenen Knotentypen bzw. Interfaces, ihre Namen (erhält man durch *getNodeName()*) und ihren Wert (*getNodeValue()*).

Interfaces	**Name**	**Wert**
Attr	Name des Attributs	Wert des Attributs
CDATASection	"#cdata-section"	Inhalt der CDATA-Section
Comment	"#comment"	Inhalt des Kommentars
Document	"#document"	null
DocumentFragment	"#document-fragment"	null
DocumentType	DokumentType Name	null
Element	Name des Tags	null
Entity	Name des Entity	null
EntityReference	Name des referenzierten Entity	null
Notation	Name der Notation	null
ProcessingInstruction	Ziel der PI	gesamter Inhalt ohne das Ziel
Text	"#text"	Inhalt des Text-Knotens

Abbildung 14: Knotentypen in DOM

Starten wir das Beispiel mit dem schon bekannten XML-Dokument *beispiel.xml,* so erhalten wir folgende Ausgabe:

Element: seminar

Element: vortrag

Attributname: thema Wert: XML-Grundlagen

Element: vortragender

Element: name

Text: Weber

endElement: name ...

Zuerst wird das erste Element mit Namen ausgegeben, darauf folgt das zweite Element und das Attribut. Die restliche Ausgabe stellt die Daten des Dokuments dar und lässt sich analog fortsetzen.

3.3 Bewertung

DOM sieht keine Möglichkeit zur Ausgabe von XML vor. Auch wenn von verschiedenen Parsern (z.B. Xerces) Klassen für die Serialisierung mitgeliefert werden, sind sie kein Bestandteil von DOM. Für Level 3 ist allerdings die Standardisierung der Serialisierung vorgesehen.

Ein großes Problem beim Arbeiten mit DOM kann der Speicherbedarf sein. Da immer das komplette Dokument eingelesen wird, bevor man damit arbeiten kann, führt dies bei großen Dateien zu Speicherproblemen, was natürlich die Performance des Systems ausbremst. Für sehr große Dokumente ist DOM also ungeeignet, wobei von vielen Parsern ein Ausweg angeboten wird. Dieser Ausweg wird "verzögertes DOM" genannt. Dabei werden die Knoten erst in den Speicher geladen, wenn sie benötigt werden.

Ein großer Vorteil von DOM ist die Möglichkeit, von einem Knoten zu seinen Kindknoten oder zum Elternknoten zu kommen, was damit zu tun hat, dass das komplette Dokument im Speicher vorliegt. Das Speichermodell hat somit Vor- und Nachteile.

Da DOM sprachunabhängig ist, hat man bei der Nutzung der Java-API den Nachteil, dass die verwendeten Konzepte nicht unbedingt objektorientiert sind, da es auch in nicht-objektorientierten Sprachen anwendbar sein soll. Die Konzepte sind damit für die Java-Welt teilweise nicht optimal. Diesen Nachteil werden wir im nächsten Kapitel bei der Besprechung von JDOM nicht sehen.

4 JDOM

4.1 Allgemeines zu JDOM

JDOM [7] ist eine Java-API, welche es ermöglicht auf XML-Dokumente zuzugreifen. Dies geschieht mit Hilfe eines Baummodells. Es gibt noch keine endgültige Version von JDOM. In dieser Seminararbeit wird die Beta 8-Version von JDOM beschrieben. JDOM ist kein Parser, sondern eine XML-Dokumentdarstellung in Baumstruktur. Die Kernklassen von JDOM sind in Abbildung 15 dargestellt.

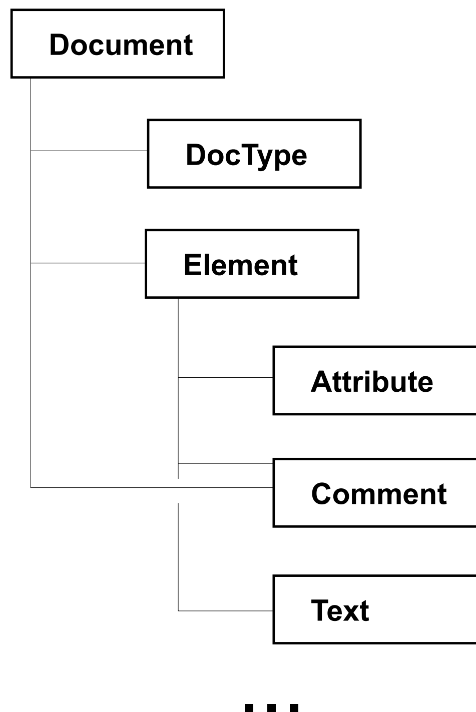

Abbildung 15: JDOM-Kernklassen

Den Kern bildet die Klasse *org.jdom.Document*. Diese Klasse repräsentiert ein XML-Dokument und dient als Container für alle anderen JDOM Strukturen. *Element* repräsentiert ein XML-Element, *Attribute* ein Attribut usw.

JDOM interoperiert eng mit SAX und DOM. Es ist jedoch keine Abstraktionsschicht oder Erweiterung von DOM oder SAX. Um ein XML-Dokument in JDOM einzulesen, steht das Package *org.jdom.input* zur Verfügung. Dort sind zwei Builder-Klassen vorhanden: *SAXBuilder* und *DOMBuilder*. Zum Einlesen von Daten aus Streams oder Dateien eignet sich der *SAXBuilder* am besten.

```
SAXBuilder builder = new SAXBuilder();
Document doc = builder.build(new FileInputStream("beispiel.xml"));
```

Abbildung 16: *SAXBuilder*

In Abbildung 16 wird zuerst ein SAXBuilder erzeugt und mit dessen Hilfe ein *org.jdom.Document*-Objekt erstellt. Dieses Objekt stellt den Wurzelknoten des eingelesenen Dokuments dar. Ist das XML-Dokument bereits in einer DOM-Struktur, so verwendet man den *DOMBuilder*. Bei Streams oder Dateien eignet sich der DOM-Builder nicht. Dies wäre ineffizient, da zuerst ein DOM-Baum erzeugt werden muss, welcher in JDOM konvertiert wird. Der DOM-Baum wird von vielen Parsern mit Hilfe von SAX erzeugt, wodurch die Erzeugung des DOM-Baums ein unnötiger Zwischenschritt bedeutet. Der *DOMBuilder* führt die Konvertierung von der DOM- zur JDOM-API durch, was man in Abbildung 17 sieht.

```
DOMBuilder builder = new DOMBuilder();
Document doc = builder.build(DOMDocumentObject);
```

Abbildung 17: *DOMBuilder*

Es wird ein *DOMBuilder* erzeugt und das vorhandene *DOMDocumentObject* in ein *org.jdom.Document* umgewandelt. Dies geschieht mit Hilfe der Methode *build.*

Für die Ausgabe von XML stellt JDOM das Package *org.jdom.output* bereit. Dieses Package enthält die drei Klassen *DOMOutputter, SAXOutputter* und *XMLOutputter.* Der *DOMOutputter* überführt die JDOM-Strukturen in DOM-Strukturen. Wie dies funktioniert, ist in Abbildung 18 dargestellt.

```
DOMOutputter outputter = new DOMOutputter();
org.w3c.dom.Document doc = outputter.output(JDOMDocumentObject);
```

Abbildung 18: *DOMOutputter*

Das Erzeugen von SAX-Ereignissen aus einem JDOM-Dokument funktioniert auf ähnliche Art und Weise:

```
SAXOutputter outputter = new SAXOutputter();
outputter.setContentHandler(myContentHandler);
outputter.setErrorHandler(myErrorHandler);
outputter.output(JDOMDocumentObject);
```

Abbildung 19: *SAXOutputter*

Der Umgang mit dem SAXOutputter funktioniert analog zu normalen SAX-Ereignissen. Die Handler werden bei dem *SAXOutputter*-Objekt registriert, und durch den Aufruf der Methode *output* werden Ereignisse für die Handler abgeschickt.

Die wichtigste Klasse zur Ausgabe ist die Klasse *XMLOutputter.* Sie gibt XML in einen *OutputStream* oder *Writer* aus. Damit ist es auf einfache Weise möglich, XML in eine Datei zu schreiben.

```
XMLOutputter outputter = new XMLOutputter();
FileOutputStream stream = new FileOutputStream(“ergebnis.xml”);
outputter.output(JDOMDocumentObject, stream);
```

Abbildung 20: *XMLOutputter*

In Abbildung 20 wird ein *XMLOutputter* und ein *FileOutputStream* erzeugt. Durch den Aufruf der Methode *output* wird das *JDOMDocumentObject* mit Hilfe des FileOutputStreams in die Datei ergebnis.xml geschrieben. Dies erzeugt eine XML-Datei ohne Einrückungen oder Zeilenumbrüche. Um dies zu erreichen, gibt es noch andere Konstruktoren, wie zum Beispiel:

```
XMLOutputter(String indent);
XMLOutputter(String indent, boolean newlines);
```

Abbildung 21: verschiedene Versionen von *XMLOutputter*

Der erste Konstruktor erzeugt einen *XMLOutputter,* bei dem die angegebene Einrückung genutzt wird. Allerdings werden keine Zeilenumbrüche hinzugefügt. Beim zweiten Konstruktor kann dies im übergebenen *boolean*-Wert angegeben werden.

4.2 Unterschiede zu DOM

Als erstes gilt es hier zu festzustellen, dass JDOM keine Erweiterung oder die Java-Version von DOM ist. Es ist eine eigene Java-API für die XML-Darstellung. Da JDOM auf Java beschränkt ist, können dort im Gegensatz zu DOM die Java-Collection-Klassen genutzt werden. Eine Liste von Knoten wird in DOM mit Hilfe des Interfaces *NamedNodeMap* oder *NodeList* verwaltet. In JDOM kann für solche Aufgaben auf die Standard-Java-Klassen zurückgegriffen werden. Dies sind *java.util.List, java.util.Map,* usw. Dies hat für Java-Entwickler den Vorteil, dass die gewohnte API genutzt werden kann.

Ein weiterer Unterschied besteht darin, dass JDOM aus konkreten Klassen besteht. So können die Klassen *Document, Element, Attribute,* usw. direkt instanziiert werden. Die Erzeugung eines neuen JDOM-Dokuments ist in Abbildung 22 beschrieben:

```
Element rootElement = new Element("root");
Document doc = new Document(rootElement);
```

Abbildung 22: Erzeugung eines JDOM-Dokuments

Bei DOM geschieht die Erzeugung eines neuen Dokuments mit Hilfe einer *Factory.* In DOM sind *Document, Element, usw.* keine Klassen, sondern nur Interfaces. In JDOM gibt es auch die Möglichkeit der Nutzung der Klasse *JDOMFactory.*

Ein Vorteil von JDOM gegenüber DOM ist der Ausgabemechanismus. In JDOM ist es mit Hilfe der Klasse *XMLOutputter* möglich, auf einfache Weise eine neue XML-Datei zu schreiben. In DOM benötigt man dazu eine externe Klasse. Ein Nachteil von JDOM ist, dass es, im Gegensatz zu DOM, noch nicht standardisiert ist. JDOM durchläuft gerade einen Standardisierungsprozess vom Java Community Process (JCP), welcher offiziell als Java Spezification Request 102 (JSR-102) [12] bezeichnet wird. JDOM soll auch in die nächste Version von JAXP aufgenommen werden, welches wir im nächsten Kapitel behandeln.

5 Java API for XML Processing (JAXP)

5.1 Allgemeines zu JAXP

JAXP [8] ist eine von Sun entwickelte Java-API. Ihre Hauptaufgabe besteht darin, SAX und DOM herstellerunabhängig (unabhängig vom Parser) nutzen zu können. Durch eine Abstraktionsschicht erlaubt es JAXP den Parser zu wechseln, ohne den Quellcode zu verändern. Durch das Wechseln der JAXP-Distribution kann man auch den verwendeten Parser wechseln. Die verschiedenen Distributionen werden mit unterschiedlichen Parsern ausgeliefert. Die Version von Sun arbeitet mit Crimson, die von Apache mit Xerces. Zum Wechseln des Parsers muss nur die Klassenpfadeinstellung oder die JAXP-Systemeigenschaft (jaxp.properties) geändert werden. Der

Code muss nicht neukompiliert werden. In der JAXP-Systemeigenschaft kann eingestellt werden, welche Parser-Factory verwendet wird. In Abbildung 22 wird die *SaxParserFactory* auf den Parser Xerces eingestellt.

```
javax.xml.parsers.SAXParserFactory =
          org.apache.xerces.XercesFactory;
```

Abbildung 23: *jaxp.properties*

Ein weiterer Teil von JAXP ist die *Transformation API for XML* (TrAX). TrAX ermöglicht eine herstellerneutrale XML-Dokumenttransformation mittels XSL Prozessoren. Dies wird durch das Package *javax.xml.transform* und seine Unterpackages erreicht.

5.2 Beispiele zur Verwendung von SAX und DOM

Im Folgenden werden wir uns genauer anschauen, wie man mit JAXP auf SAX bzw. DOM zugreifen kann. Beginnen wir mit SAX.

```
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setValdating(true);
SAXParser parser = factory.newSAXParser();
```

Abbildung 24: *SAXParser*

In Abbildung 24 erzeugen wir zuerst ein *javax.xml.parsers.SAXParserFactory*-Objekt. Dann schalten wir die Validierung ein. Durch die Methode *setFeature* kann man noch weitere Einstellungen für den zu erzeugenden Parser vornehmen. Diese Methode funktioniert analog zu der *setFeature*-Methode der Klasse *org.xml.sax.XMLReader*, welche in Kapitel 2 besprochen wurde. Diese Konfigurationsoptionen betreffen sämtliche Parser-Instanzen, die wir von der Factory erhalten. Durch den Aufruf der Methode *newSAXParser* erhalten wir eine Instanz der Klasse *javax.xml.parsers.SAXParser*. Mit diesem *SAXParser* sind wir nun in der Lage, mit dem Parsen zu beginnen:

```
parser.parse(“beispiel.xml”,new MeinHandler());
```

Abbildung 25: Starten des Parsers

Der Methode *parse* wird ein String mit der Adresse der zu parsenden Datei und eine Instanz von *MeinHandler* übergeben. Die Klasse *MeinHandler* muss noch implementiert werden. Sie erbt von *org.xml.sax.helpers.DefaultHandler*. Diese Klasse stellt eine Standardimplementierung verschiedener SAX-Handler bereit. Darunter befinden sich auch der *ContentHandler* und der *ErrorHandler*. Wir überschreiben einige Handler-Methoden, um die eingelesenen Daten wieder auszugeben. Dies geschieht analog zu Kapitel 2.2 und wird hier nicht weiter erläutert.

Von der *SAXParserFactory* kann man sich eine Instanz von *org.xml.sax.XMLReader* erzeugen lassen. Dies geschieht mit der Methode *getXMLReader*. Mit dieser Instanz von *XMLReader* stehen die üblichen SAX-Methoden zur Verfügung.

Um mit Hilfe von JAXP mit DOM zu arbeiten, muss man zuerst eine *DocumentBuilderFactory* erzeugen. Auch hier kann man mit der Methode *setValidating* die Validierung ein- und ausschalten. Die *setFeature*-Methode der *SAXParserFactory* existiert allerdings nicht. Weitere Optionen können über einzelne Methoden gesetzt werden, wie beispielsweise das Ignorieren von Kommentaren *(setIgnoringComments)* oder das Ignorieren von Leerzeichen *(setIgnoringElementContentWhitespace)*. Durch die Methode *newDocumentBuilder* erhält man einen *javax.xml.parsers.DocumentBuilder.*

```
DocumentBuilderFactory factory =
        DocumentBuilderFactory.newInstance();
factory.setValidating(true);
DocumentBuilder builder = factory.newDocumentBuilder();
```

Abbildung 26: *DocumentBuilder*

Mit dem DocumentBuilder können wir nun ein Dokument parsen, was uns ein *org.w3c.dom.Document* zurückliefert.

```
Document doc = builder.parse("beispiel.xml");
```

Abbildung 27: Parsen von *beispiel.xml*

Der Methode *parse* wird hier ein String mit der URI des zu parsenden Dokuments übergeben. Es gibt noch andere Varianten dieser Methode, welche einen *InputStream*, ein *File*-Objekt oder eine *InputSource* entgegennehmen. Diese Methoden geben uns ein *Document*-Objekt zurück, mit dem wir weiter verfahren können, wie es schon in Kapitel 3 beschrieben wurde.

6 XL

6.1 Allgemeines zu XL

XL [9] ist ein Forschungsprojekt der TU München [10], in dem eine XML-Programmiersprache entwickelt wird. Mit XL soll es möglich sein, Web Services ohne die Nutzung anderer Programmiersprachen implementieren zu können. Bei üblichen, mittels Java entwickelten, XML-Webapplikationen gibt es zwei Schwierigkeiten:

1. Die XML-Daten müssen in Java-Objekte umgewandelt werden, bevor mit ihnen gearbeitet werden kann. Zum Ende der Verarbeitung müssen die Java-Objekte wieder in XML-Daten konvertiert werden.

2. Java-Objekte müssen umgewandelt werden, um die Daten mittels JDBC in eine relationalen Datenbank zu speichern.

In Webapplikationen werden oft 80 Prozent des Codes für die Umwandlung von Daten beansprucht. In XL soll dies verhindert werden. Eine schnelle Entwicklung von Web Services soll ermöglicht werden. In XL steht XML als einzige Datenrepräsentation und einziges Typsystem zur Verfügung. Die Syntax von XL ist jedoch

nicht an XML angelehnt. Eine weitere wichtige Eigenschaft von XL ist, dass andere semantische Aktionen, wie Datenvalidierung oder Fehlerbehandlung, so gut wie möglich von der Applikationslogik getrennt werden. Bei der Programmierung mit XL soll sich der Programmierer auf die Applikationslogik konzentrieren können. Der Programmierer soll sich keine Gedanken über Performance, Internet- oder Datenbankprotokolle machen. Diese Aspekte sollen auf automatischem Weg gelöst werden. Weiterhin soll XL mit den schon existierenden (Java basierten) Web Services und deren Infrastruktur koexistieren. Im Folgenden wird eine Auswahl spezifischer Anforderungen [11] aufgelistet, welche bei der Entwicklung von XL berücksichtigt werden:

1. XL hält die XML-W3C Standards (XML Schema, XQuery, XPath, XSLT, XML Forms, XML Protocol) ein.
2. Die Sprache muss Konstrukte für die Implementierung von Geschäftsprozessen bereithalten.
3. Die Komposition von komplexen Diensten aus Basisdiensten muss erlaubt sein. Auch die Einbindung von nicht-XL Diensten muss transparent ermöglicht werden.
4. Ein XL-Web Service kommuniziert mit anderen Web Services mittels (XML-) Nachrichten.
5. Die Sprache soll plattformunabhängig sein. Es soll möglich sein, Programme auf jedem Rechner virtuell ablaufen zu lassen, wenn der Rechner einen Interpreter für die Sprache unterstützt.
6. Programmierer sollen nur Kenntnis von der logischen Struktur der XML-Daten haben. Die physikalische Struktur ist für sie uninteressant.
7. Die Sprache soll es erlauben, eine Sequenz von Aktionen isoliert und atomar auszuführen (Transaktionen).
8. Es muss möglich sein, Zugangskontrolle und die eingeschränkte Nutzung eines Dienstes zu implementieren.

6.2 Beispiel

Als Beispiel werden wir ein Fragment eines webbasierten Auktionssystems in XL betrachten. Es wird die Registrierung eines neuen Bieters beschrieben.

```
service HTTP://www.auktion.com
 ...
 let bieter;
 ...
 operation registriereBieter
   precondition exists($input/uri);
   precondition empty($bieter[@uri = $input/uri]);
   postcondition exists($bieter[@uri = $input/uri]);
  body
   insert $input into $bieter;
   let $output:=<message>Danke für die Registrierung.</message>
   endbody
 endoperation
...
endservice
```

Abbildung 27: Auktionssystem

Zu Beginn des Auktionssystems wird die Adresse unseres Systems als *http://www.auktion.com* spezifiziert. Im Folgenden wird die interne Variable *bieter* definiert. Auf diese Variable kann mit *$bieter* zugegriffen werden. Die Operation *registriereBieter* hat zwei Bedingungen, die zu Beginn der Operation überprüft werden. Zuerst wird getestet, ob der neue Bieter über eine URI erreichbar ist. Der neue Bieter wird über *$input* angesprochen. In der zweiten Vorbedingung wird getestet, ob der neue Bieter schon in den Datenbeständen vorhanden ist. Als Nachbedingung wird mittels *postcondition* spezifiziert, dass der Bieter nach der Operation in der Datenbank vorhanden sein muss. Im Hauptteil *body* wird der Bieter in die Datenbank eingefügt, und die Ausgabe wird in *$output* gespeichert.

7 Zusammenfassung

Zu Beginn dieser Seminararbeit wurde SAX beschrieben. SAX löst beim Parsen eines XML Dokuments Ereignisse aus, wodurch vom SAX Prozessor Methoden aufgerufen werden. In einem Beispiel wurden diese Methoden erläutert und die Fehlerbehandlung dargestellt.

DOM und JDOM sind baumstrukturierte Modelle für die Darstellung von XML-Dokumenten. DOM ist sprachunabhängig und die Java-Sprachbindungen von DOM wurden in einem Beispiel beschrieben. Von dem für Java entwickelten JDOM wurde der Ein- und Ausgabemechanismus für XML-Dokumente erklärt.

In Kapitel 5 wurde gezeigt, wie man mit Hilfe von JAXP parserunabhängig mit SAX und DOM arbeiten kann und den Parser ohne eine Neukompilierung des Codes wechseln kann.

XL ist eine in der Entwicklung befindliche XML-Programmiersprache für Web Services. Hier wurden die grundlegenden Anforderungen an XL beschrieben, welche bei der Entwicklung berücksichtigt werden.

8 Literatur

[1] Brett McLaughlin, Sascha Kerksen, Jürgen Key: Java & XML, deutsche Übersetzung, 2. Auflage

[2] Apache XML Project, http://xml.apache.org

[3] Document Object Model Level2 Core Specification, W3C (http://www.w3.org), 13. November 2000

[4] XML Parser for Java, http://alphaworks.ibm.com/tech/xml4j

[5] Oracle XML Homepage, http://technet.oracle.com/tech/xml

[6] Crimson, Java XML Parser, http://xml.apache.org/crimson

[7] JDOM Homepage, http://www.jdom.org

[8] Java API for XML Processing Version 1.2 Final Release, 6. September 2002, Sun Microsystems Inc. http://java.sun.com/xml/jaxp/index.html

[9] Daniela Florescu, Andreas Grünhagen, Donald Kossmann. XL: AN XML Programming Language, Technical Report, December 2001.

[10] XL: An XML Programming Language, http://xl.in.tum.de

[11] Daniela Florescu, Andreas Grünhagen, Donald Kossmann. XL: An XML Programming Language for Web Service Specification and Composition. WWW2002, International World Wide Web Conference, Honolulu, USA, May 7-11, 2002

[12] Java Community Process, JSR-102, http://java.sun.com/aboutJava/communityprocess/jsr/jsr_102_jdom.html