**In diesem Kapitel**
- Einleitung
- Die Entstehung des DOM
- Die Baumstruktur von DOM
  - *Document, Element, Attribute, Attribute, Leaf*
- DOM und Java
  - *JAXP*
  - *DOM3*
  - *Das Node Interface*
  - *Das NodeList Interface*
  - *DOM Exceptions*
- DOM oder SAX?
- Zusammenfassung

# 9. *DOM - Document Object Model*

## *9.1. Einleitung*

Das Document Object Modell (DOM) ist eine abstrakte Datenstruktur, mit deren Hilfe XML Dokumente dargestellt werden können, in Form eines Baumes. W3C hat verschiedene Interfaces definiert, zusammengefasst im Package

**`org.w3c.dom`** in
**`http://www.w3.org/TR/DOM-Level-2-Core/Overview.htmlorg.w3c.dom`**

Diese Interfaces umfassen Elemente, Attribute, PCDATA (Parsed Character Data), Kommentare und verarbeitungsanweisungen (Processing Instruction).

Zm Vorgehen:
- zuerst beschreiben wir die Struktur (Interfaces von W3C)
  zum Teil illustrieren wir diese anhand von Java Interfaces
- dann schauen wir uns die Java Implementierung an
- am Schluss gehen wir auf spezielle Fragen ein.

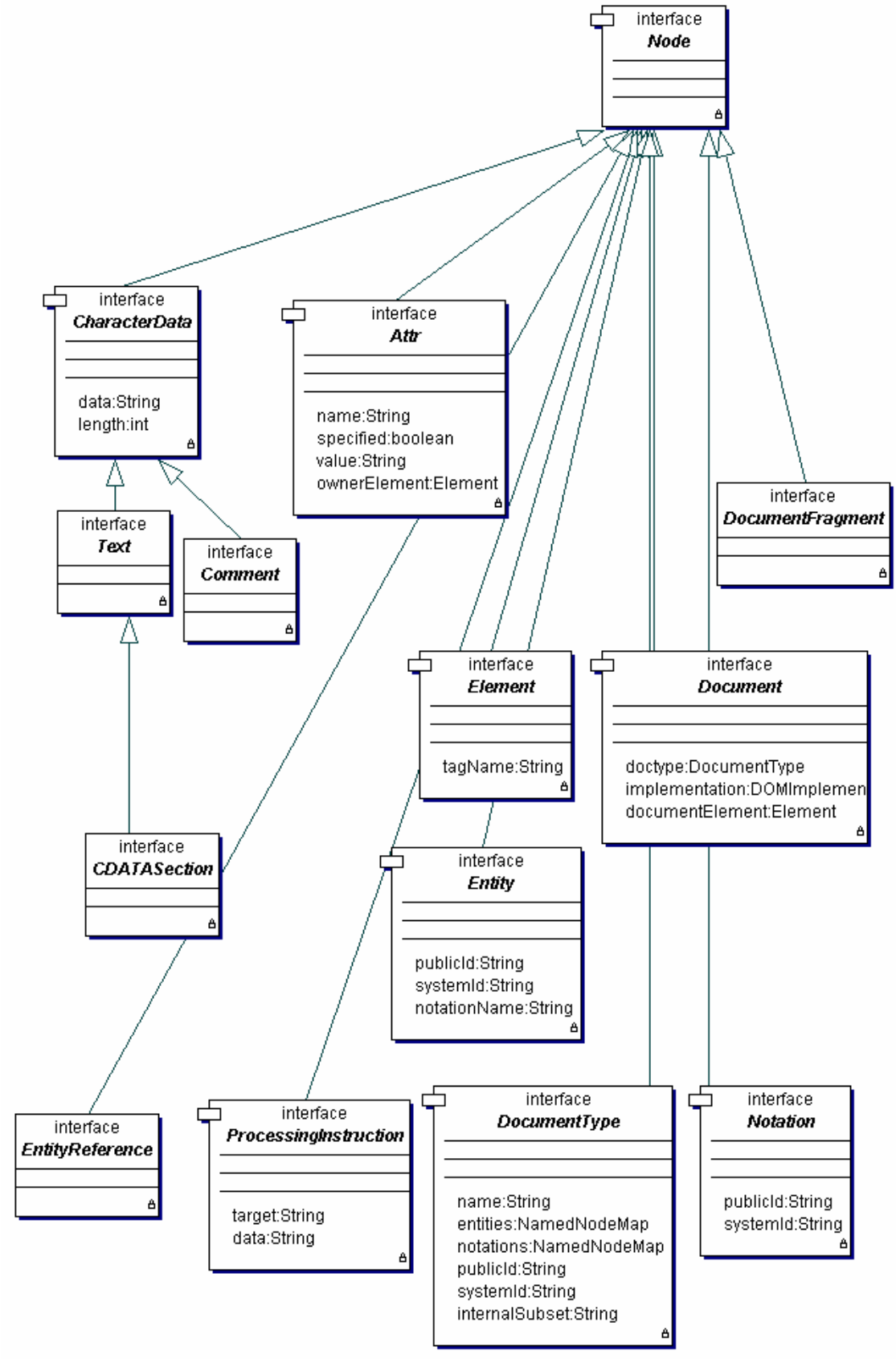
Als Parser verwenden wir
- Crimson (Apache, Sun: dieser ist im J2SD ab Version 1.4)
- Xerces (Apache, IBM: dieser muss separat heruntergeladen und in den Classpath aufgenommen werden).

Die Funktionalitäten der beiden Parser sind recht unterschiedlich.

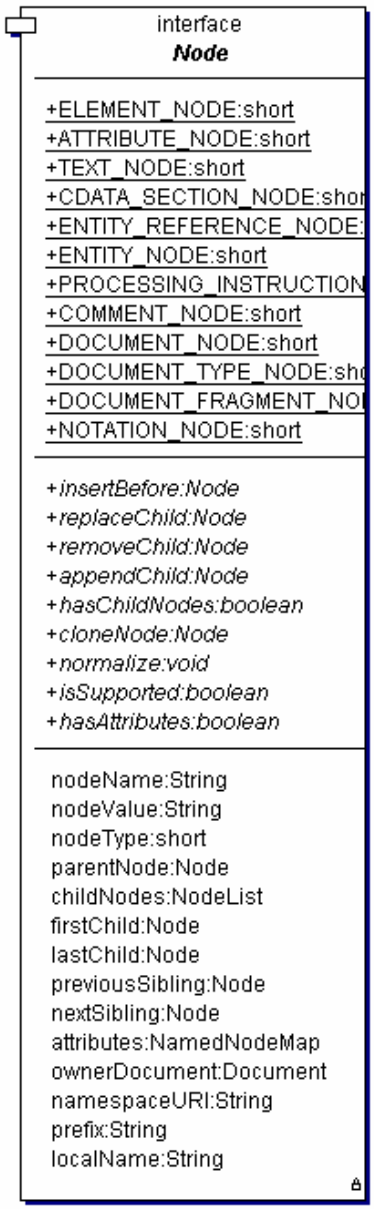
Neben diesen beiden Parsern existieren einige weitere, kommerzielle DOM Parser, beispielsweise von der Firma Oracle.

Eine vereinfachte Java DOM Implementierung wird von JDOM.org angeboten.
Dieser Parser arbeitet konsequent mit Klassen, nicht mit Interfaces, wie die meisten andern Java DOM Parsern. Sie finden einen ersten Überblick über DOM Parser in Modul 5 (XML Parser).

Zu W3C:
Alle Interfaces sind Sub-Interfaces von **`Node`**. Dieses enthält grundlegende Methoden, welche für die Navigation und die Bearbeitung des DOM Baumes eingesetzt werden können.



Das Node Interface ist sehr umfangreich:




Die (final) Attribute (Konstanten) definieren die unterschiedlichen Knotentypen mithilfe numerischer Werte:

- <Knotentyp>_NODE

Die Methoden gestatten

- den Zugriff, das Ersetzen, Einfügen oder Löschen von Knoten.

Abfragen bestimmter Eigenschaften:

- hat der Knoten Subknoten?
- besitzt der Knoten Attribute?

Schliesslich werden unterschiedliche **`get/set`** Methoden für den Zugriff auf private Attribute verlangt:

- die Beschreibung des Knotens (**`nodeName`**)

Dieser Block entspricht (in Together Terminologie) „Properties“, also privaten Attributen, auf die mithilfe von **`get/set`**-Methoden zugegriffen werden kann.

Die Wurzel des XML Dokuments ist ein **`Document`** Objekt. Dieses repräsentiert ein wohlgeformtes Dokument. Der DOM Parser kann ein XML Dokument aus einem Stream lesen und ein **`Document`** Objekt bilden.

Ein Client Programm kann mithilfe der Document Methoden und den andern DOM Interfaces-Methoden im DOM Baum navigieren und den Baum manipulieren bzw. Daten aus dem Baum extrahieren.

Dabei wird der DOM Baum im Speicher belassen und nicht serialisiert! Das hat als Konsequenz, dass bei DOM viel Speicher benötigt wird; die eigentliche Manipulation erfolgt aber entsprechend schnell.

Ein DOM Baum kann auch im Memory aufgebaut und dann als Dokument ausgelagert, d.h. auf die Harddisk geschrieben werden.

DOM selber wurde mithilfe der Interface Definition Language der OMG (Object Management Group: OMG.org) völlig sprachneutral definiert. Konkrete DOM „Bindings“, also Abbildungen (Beschreibungen) bestehen für unterschiedliche Programmiersprachen, wie etwa JavaScript, Java, C++, Perl, ...

Beispiele zu DOM mit JavaScript finden Sie im XML Skript.

## 9.2. Die Entstehung von DOM

Zuerst entstand das inoffizielle DOM Modell, nun als Level-0 bezeichnet. Dieses wurde in unterschiedlichen Browsern unterschiedlich implementiert. Das Modell konnte auch mit JavaScript genutzt werden.

Die erste W3C Version wurde als DOM Level 1 bekannt, kurz DOM1. Dieser Level ist auch heute noch aktuell: er bietet das, was ein Durchschnittsprogrammierer von DOM erwartet.

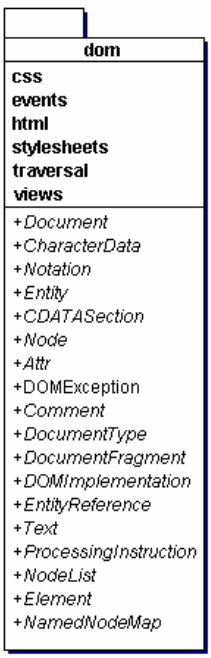
In DOM Level 2 wurde zuerst Level 1 bereinigt und Namespaces hinzugefügt (Interfaces **`Element`** und **`Attr`**) und weitere Interfaces hinzugefügt. Level 2 ist nun Standard.

Zurzeit wird an der Definition (und Implementierung) von Level 2 gearbeitet. Level 3 bereinigt wieder einige Probleme aus Level 2 und fügt weitere Interfaces hinzu. Insbesondere soll das Behandeln von XML Dokumenten, und das Generieren eines DOM Objekts daraus, verbessert werden.

In der Regel sollten die Level aufwärts kompatibel sein: Level 2 sollte später auch mit Level 3 Parsern bearbeitbar sein.

## 9.3. DOM Sub-Packages

DOM besteht aus mehreren Packages, wie man im folgenden UML Diagramm erkennen kann:




Core: **`org.W3c.dom`**
Die Interfaces und Klassen aus diesem Package stellen eine Basisfunktionalität sicher.

CSS: **`org.w3c.dom.css`**
In diesem Package werden Interfaces zusammengefasst, welche sich mit den Cascading Stylesheets befassen:

- **`CSSStyleSheet`**
- **`CSSRuleList`**
- **`CSSRule`**
- **`CSSStyleRule`**
- `...`

Events: **`org.w3c.dom.events`**
Diese Interfaces beschreiben auf generische Art und Weise ein System zur Implementierung von Event Listener, welche einem Knoten hinzugefügt werden können.

Es gibt unterschiedliche Event Kategorien:

- **UIEvents**
- **MouseEvents**
- **MutationEvents**
- **HTMLEvents**

HTML: **org.w3c.dom.html**
Darin befinden sich Interfaces, mit deren Hilfe HTML-Teile beschrieben werden können:

- **HTMLHtmlElement**
- **HTMLHeadElement**
- **HTMLParagrapgElement**

Stylesheets: **org.w3c.dom.stylesheets**

Diese Interfaces werden zur Darstellung von Stylesheets benötigt:

- **StyleSheet**
- **StyleSheetList**
- **MediaList**
- **LinkStyle**
- **DocumentStyle**

Traversal: **org.w3c.dom.traversal**

Dieses Package umfasst Hilfsklassen, beispielsweise um einen Baum zu durchlaufen, Knoten auszufiltern, welche bestimmte Bedingungen erfüllen.

Views: **org.w3c.dom.views**
Die zwei Interfaces **AbstractView** und **DocumentView** können zur Definition unterschiedlicher Sichten, Views, auf ein Dokument benutzt werden. Die Interfaces werden aber durch Parser in der Regel nicht unterstützt.

Je nach Parser werden mehr oder weniger dieser Packages unterstützt. Die Methode hasFeature() kann benutzt werden, um abzufragen, ob ein bestimmtes Feature zur Verfügung steht (siehe weiter unten).

```
public boolean hasFeature(String name, String version)
```

Die konkrete Implementierung der Interfaces kann sich von einer andern wesentlich unterscheiden.

**Beispiel**
Xerces:

```
org.apache.xerces.dom.DOMImplementationImpl
                                        implements DOMImplementation
```

Oracle:

```
XMLDOMImplementation implements DOMImplementation
```

Natürlich sind die Klassen selber auch unterschiedlich ausgestattet (Konstruktoren, …).

**Beispiel**
Abfrage der Features der beiden Parser Crimson und Xerces

```
package crimson;
import org.apache.crimson.tree.DOMImplementationImpl;
import org.w3c.dom.DOMImplementation;

public class CrimsonParserFeatures {

	public static void main(String[] args) {

		// parser abhängig
		DOMImplementation implementation = new DOMImplementationImpl();
		String[] features =
			{
				"Core","XML","HTML","Views","StyleSheets",
				"CSS","CSS2",
				"Events","UIEvents","MouseEvents","MutationEvents",
				"HTMLEvents",
				"Traversal",
				"Range" };
		for (int i = 0; i < features.length; i++) {
			if (implementation.hasFeature(features[i], "2.0")) {
				System.out.println("Crimson unterstützt: " +
							features[i]);
			} else {
				System.out.println("Crimson unterstützt nicht: " +
						features[i]);
			}
		}
	}
}
```

Wenn wir die Features des Xerces Parsers abfragen woollen, müssen wir lediglich die Imports anpassen:

```
import org.apache.xerces.dom.DOMImplementationImpl;
import org.w3c.dom.DOMImplementation;
```

Als Ausgabe erhalten wir mit dem Xerces Parser:

```
Xerces unterstützt: Core
Xerces unterstützt: XML
Xerces unterstützt nicht: HTML
Xerces unterstützt nicht: Views
Xerces unterstützt nicht: StyleSheets
Xerces unterstützt nicht: CSS
Xerces unterstützt nicht: CSS2
Xerces unterstützt: Events
Xerces unterstützt nicht: UIEvents
Xerces unterstützt nicht: MouseEvents
Xerces unterstützt: MutationEvents
Xerces unterstützt nicht: HTMLEvents
Xerces unterstützt: Traversal
Xerces unterstützt: Range
```

Mit Crimson:

```
Crimson unterstützt: Core
Crimson unterstützt: XML
```

```
Crimson unterstützt nicht: HTML
Crimson unterstützt nicht: Views
Crimson unterstützt nicht: StyleSheets
Crimson unterstützt nicht: CSS
Crimson unterstützt nicht: CSS2
Crimson unterstützt nicht: Events
Crimson unterstützt nicht: UIEvents
Crimson unterstützt nicht: MouseEvents
Crimson unterstützt nicht: MutationEvents
Crimson unterstützt nicht: HTMLEvents
Crimson unterstützt nicht: Traversal
Crimson unterstützt nicht: Range
```

Wie sich Ihr *spezifischer Parser* verhält, können Sie ja nun leicht testen:

- suchen Sie die Klasse, welche **`DOMImplementation`** implementiert
- prüfen Sie, ob die Methode **`hasFeature`**() implementiert wird
- dann können Sie die obigen Features und mehr testen!

### *9.4. DOM als Baum*

DOM basiert auf einem Datenmodell, welche sich an jenen von andern XML Datenmodellen leicht unterscheidet:

- ein DOM Dokument besteht aus Knoten unterschiedlichen Typs, die zusammen einen Baum formen
- der Baum besitzt genau eine Wurzel
- jeder Knoten im Baum besitzt genau einen übergeordneten Knoten (ausser der Wurzel).
- Jeder Knoten besitzt eine Liste von Kind-Knoten, welche auch leer sein kann. In diesem Fall nennt man den Knoten einen Blattknoten.

Es gibt auch Knoten, welche nicht Teil der Baumstruktur sind:

- jeder Attribut-Knoten gehört zu einem Element-Knoten. Man betrachtet ihn aber nicht als Teil des Baumes.
- Entities sind zwar Knoten, aber sie sind nicht im DOM Baum eingebaut.

Ein DOM Dokument besteht also aus Baum-Knoten, Knoten, welche mit dem Baum zu tun haben und nicht zusammenhängenden Knoten.

Jeder *Knoten* besitzt

- einen lokalen Namen
- eine Namensraum URI
- einen Präfix, der aber auch leer sein kann,

Der Name der Knoten, der *Knotenname* (node name):

- für *Elemente* und *Attribute* ist dies *der Präfixed-Name*
- für alles andere (*Entities, Notations*) ist dies einfach der *Name des Objekts*
- bei *namenlose Knoten*, wie beispielsweise Textknoten, ist der Knotenname abhängig vom Knotentyp:
  - #document
  - #comment
  - #text
  - #cdata-section
  - #document-fragment

Zudem besitzt jeder Knoten einen String Wert:

- bei textähnlichen Knoten, wie Textknoten oder Kommentaren, ist dies in der Regel der Text des Knotens
- bei Attributen handelt es sich um einen normalisierten Wert des Attributs
- für alle andern Knoten ist das Feld leer

DOM teilt die Knoten in mehrere *Typen* ein:

1. **Dokument Knoten**
2. **Element Knoten**
3. **Text Knoten**
4. **Attribut Knoten**
5. **Verabeitungsanweisung Knoten (Processing Instruction)**
6. **Kommentar Knoten**
7. **Dokumenttyp Knoten**
8. Dokumentfragment Knoten
9. Notationsknoten
10. CDATA Sektionsknoten
11. Entity Knoten
12. Entity Referenzknoten

Die ersten sieben Knoten werden am meisten eingesetzt. Die andern eher selten.

### 9.4.1. Dokument Knoten

Jeder DOM *Baum* besitzt genau einen Wurzel Knoten, ein Root, den *Document* Knoten.

- Der Dokument Knoten besitzt **genau** einen *Element* Child Knoten.
- Falls das Dokument einen Dokumenttyp Knoten besitzt, dann ist auch dieser ein Child Knoten.
- Auch Kommentare können ganz zuoberst angehängt werden.
- Schliesslich können sich noch Procesing Instructions auf dieser obersten Ebene befinden.

**Beispiel**

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/css" href="xml-rpc.css"?>
<!--xml-stylesheet Verarbeitung  in einem
    XML-RPC Dokument ist kaum sinnvoll, aber gestattet
   In  SOAP wären Processing Instructions verboten. -->
<!DOCTYPE methodCall SYSTEM " Kap9_Beispiel_1.dtd">
<methodCall>
    <methodName>getPrime</methodName>
    <params>
        <param>
            <value>
                <int>4311</int>
            </value>
        </param>
    </params>
</methodCall>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSPY v2004 rel. 2 U (http://www.xmlspy.com) -->
<!--DTD generated by XMLSPY v2004 rel. 2 U (http://www.xmlspy.com)-->
<!ELEMENT methodCall (methodName, params)>
<!ELEMENT methodName (#PCDATA)>

<!ELEMENT params (param)>
<!ELEMENT param (value)>

<!ELEMENT value (int)>
<!ELEMENT int (#PCDATA)>
```

Unser Beispiel besitzt folgende Struktur, ausgehend vom Dokument Wurzelknoten:

- Einen *Processing Instruction* Knoten für
  <?xml-stylesheet type="text/css" href="xml-rpc.css"?>
- Einen *Kommentar* Knoten
  <!--xml-stylesheet Verarbeitung in einem
  XML-RPC Dokument ist kaum sinnvoll, aber gestattet
  In SOAP wären Processing Instructions verboten. -->
- Einen *Document Type Definitionsknoten*
  <!DOCTYPE methodCall SYSTEM "Kap9_Beispiel_1.dtd">
- Und schlieslich den *Element* Knoten (davon darf es nur genau einen als Child Knoten des Wurzelknotens geben, sonst ist das Dokument nicht wohlgeformt).
  <methodCall>...</methodCall>

Die eigentliche XML Deklaration (inklusive Version, Encoding und Standalone Information) und alle Whitespaces (Leerzeichen, Tabs, CR, LF) werden nicht als Knoten des Baumes gemäss W3C angesehen. Der Parser baut diese nicht in den DOM Baum ein.

### 9.4.2. Element Knoten

Jeder *Element* Knoten besitzt

- einen local Name
- eine Namespace URI (kann auch null sein, falls das Element in keinen NS gehört))
- einen Präfix (auch dieser kann null sein)
- eventuell Kinder Knoten

**Beispiel**

```
<value>
    <int>4311</int>
</value>
```

Als DOM Fragment:

- Der Knoten <value> besitzt den Namen value
- Er besitzt einen Child Element Knoten <int>
- *Und der <int> Child Element Knoten besitzt selber ebenfalls einen Child Knoten, den Text Knoten 4311.*

Das diente nur zum Aufwärmen. Jetzt schauen wir uns eine komplexere Situation mit Namensraum und allem drum und dran an.

**Beispiel**

```
<?xml version="1.0"?>
<!DOCTYPE methodCall SYSTEM "Kap9_Beispiel_2.dtd">
<methodCall>
    <methodName>getPrime</methodName>
    <params>
        <prim:param xmlns:prim="http://all-primes.org" xmlns="http://www.w3c.org/NameSpace">
            Auch das noch: Text im <zusatz>Methodenaufruf</zusatz> mit dem Wert
            <value>
                <int>4311</int>
            </value>
        </prim:param>
    </params>
</methodCall>
```

Als DOM Fragment besitzt der Knoten *<prim:param >*
- den Namen *prim:param*
- den lokalen Namen *param*
- den Präfix *prim*
- den Namspace *http://all-primes.org*
- die Child Knoten:
  - Text Knoten : *Auch das noch: Text im*
  - Element Knoten *<zusatz>*
    mit lokalem Namen *zusatz*
    Namenspace URI *http://all-primes.org*
    Präfix *null*
  - Element Knoten *<value>*
    mit lokalem Namen *value*
    Namenspace URI *http://all-primes.org*
    Präfix *null*
  - Text Knoten *mit dem Wert*
  - Whitespace Knoten

Beachten Sie, dass selbst wenn die Whitespaces ignoriert werden, diese zu Knoten führen.

**Beispiel**
In Fragment zu Beispiel 1
```
<methodCall>
    <methodName>getPrime</methodName>
    <params>
        <param>
            <value>
                <int>4311</int>
            </value>
        </param>
    </params>
</methodCall>
```
besitzt der Knoten methodCall fünf Child Knoten:
1. einen Text Knoten, der allerdings nur Whitepaces enthält
2. den Element Knoten mit dem Namen methodName
3. einen weiteren Text Knoten bestehend aus Whitespaces
4. einen Element Knoten mit dem Namen params
5. und schliesslich wieder einen Text Knoten bestehend aus Whitespaces

Die einzelnen Knoten können ihrerseits selbst wieder Child Knoten haben.

### 9.4.3. Attribut Knoten

Ein Attribut Knoten besteht aus
- einem Namen
- einem lokalenNamen
- einem Präfix
- einem Namespace URI
- und einem String Wert

Der Wert muss gemäss XML Spezifikation normalisiert sein:
- Entity- und Zeichen- Referenzen werden aufgelöst
- White Spaces werden in ein Leerzeichen konvertiert
- Attributwerte, die nicht CDATA sind, werden folgendermassen konvertiert:
  - White Spaces werden zu Leerzeichen
  - Führende und abschliessende Leerzeichen werden abgeschnitten

Attribut Knoten können auch Child Knoten besitzen, allerdings nur Text und Entity-Referenz Knoten, welche den Wert des Attributs angeben.

Falls der Parser den DOM Baum aus einem XML Dokument aufbaut, dann werden auch *Default Werte* aus dem DTD bzw. dem XML Schema in den DOM Baum übernommen.

Attribute werden nicht als Child Knoten des Element Knotens betrachtet. Attribut Knoten bilden mit andern Knoten einen separaten Haufen von Knoten.

### 9.4.4. Blattknoten (Leaf Nodes)

Document, Element Attribut, Entity und Entity Referenz-Knoten können Child Knoten haben. Alle andern Knoten sind einfacher aufgebaut.

#### 9.4.4.1. Text Knoten

*Text* Knoten enthalten Zeichendaten des Dokuments, in Form von Zeichenketten, Strings. Spezialzeichen werden als Entities angegeben.

Der Parser versucht möglichst viel Text in einen Text Knoten zu stecken. Deswegen folgen nie zwei Text Knoten aufeinander. Sie können dies jedoch im Anwendungsprogramm erreichen, da der Parser beispielsweise beim Einfügen diese Prüfung nicht durchführt.

#### 9.4.4.2. Comment Knoten

Ein *Kommentar* Knoten besitzt :

- einen Namen (immer #comment)
- eine Zeichenkette (den Kommentar)
- einen Eltern Knoten

**Beispiel**

```
<!-- Der Mann ist das Oberhaupt der Familie, die Frau ist der Hals. Der Hals dreht den Kopf in die rechte Richtung //-->
```

Der Wert dieses Knotens ist Der Mann ist das Oberhaupt der Familie, die Frau ist der Hals. Der Hals dreht den Kopf in die rechte Richtung

Das Leerzeichen am Schluss wird mitgezählt.

#### 9.4.4.3. Processing Instruction Knoten

Ein *Processing Instruction* Knoten besitzt:

- einen Namen (das Ziel der Processing Instruction)
- eine Zeichenkette (Daten der Processing Instruction)
- einen Eltern Knoten

**Beispiel**

```
<?xml-stylesheet type="text/css" href="xml-rpc.css" ?>
```

- Der Name des Knotens ist xml-stylesheet
- Der Wert des Knotens ist type="text/css" href="xml-rpc.css"
- Das Leerzeichen zwischen dem Target und den Daten wird nicht ignoriert!
- Auch das Leerzeichen zwischen dem Wert und dem schliessenden ?>

#### 9.4.4.4. CDATA Section Knoten

Ein *CDATA Section* Knoten

- ist ein Text Knoten, der den Inhalt einer CDATA Section enthält
- Der Name des Knotens ist **`#cdata-section`**
- Der Wert des Knotens ist **`#cdata-section`**

**Beispiel**

```
<![CDATA[<?xml-stylesheet type="text/css" href="xml-rpc.css"?>]]>
```

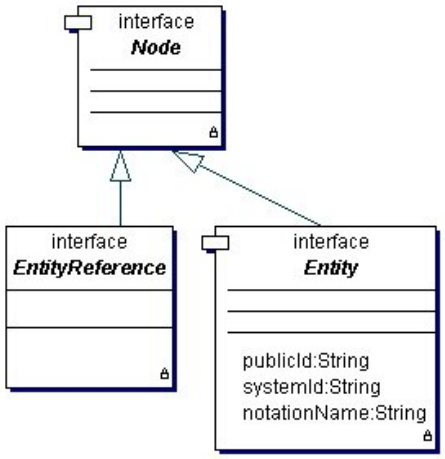
Name: #cdata-section
Wert: <?xml-stylesheet type="text/css" href="xml-rpc.css"?>

#### 9.4.4.5. Entity Reference Knoten

Wenn ein validierender Parser auf eine Entity Referenz wie **`&adresse`** stossen, dann wird er normalerweise die Referenz durch deren Wert ersetzen.

Bei nicht-validierenden Parsern ist das Ergebnis nicht voraussehbar.

Falls der Parser die Referenzen nicht auflöst, dann muss der DOM Baum die Entity Referenz als Knoten mit aufnehmen.




Jede *Entity Referenz*

- besitzt einen Namen
- in der Regel auch die Public und die System ID (Attribute im Interface)

Falls die Referenz aufgelöst wird, kann, muss aber nicht, die Entity Referenz auch als Knoten aufgenommen werden.

**Beispiel**

```
<info>&startZahl; oder 123</info>
```

Falls der Parser diese Referenz nicht auflöst, dann enthält der <info> Knoten zwei Child Knoten:

1. einen Entity Referenz Knoten &startZahl;
2. einen Text Knoten oder 123

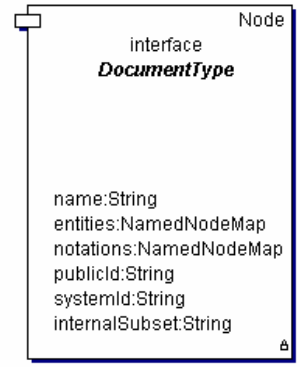
Diese Analyse ist aber nicht eindeutig:

- falls der Parser die Entity auflöst, dann könnte der Parser
  - zwischen dem <info></info> Tag als Text interpretieren
  - oder angeben, dass er einen Entity Tag (mit dem Namen startZahl; gefunden hat plus einen Textknoten mit dem Inhalt oder 123
    - in diesem Fall enthält der Entity Text einen Text Knoten mit dem Inhalt der Entity

Die Standard Referenzen **`&`**; **`<`**; **`&apos;`** und **`&quot;`** werden immer aufgelöst, d.h. der entsprechende Text (das Zeichen) wird eingefügt.

#### 9.4.4.6. Document Type Knoten

Der *Document* Type Knoten:




- besitzt einen Namen, den Namen der Document Type Deklaration für das Root Element.
- Eine public ID (diese kann auch null sein)
- Eine System ID (zwingend erforderlich)
- Ein internes DTD Subset (kann null sein)
- Ein Parent (das Dokument, in dem die Doctype Deklaration enthalten ist)
- Eine Liste von Notations
- Sowie generelle ENtities, welche in der DTD deklariert wurden.
  Der Wert eines Doctype Knotens ist immer null.

**Beispiel**

```
<!DOCTYPE mml:math PUBLIC "-//W3C//DTD MathML 2.0//EN"
 "http://www.w3.org/TR/MathML2/dtd/mathml2.dtd" [
  <!ENTITY % MATHML.prefixed "INCLUDE">
  <!ENTITY % MATHML.prefix "mml">
]>
```

(aus der XMLMath Spezifikation bei W3C)

Analyse:

Name des Knotens: **`mml:math`**
PUBLIC ID: **`"-//W3C//DTD MathML 2.0//EN"`**
SYSTEM ID: **`"http://www.w3.org/TR/MathML2/dtd/mathml2.dtd"`**
Interner DTD Subset; **`<!ENTITY % MATHML.prefixed "INCLUDE">`**
**`<!ENTITY % MATHML.prefix "mml">`**

### 9.4.5. Knoten, die nicht zum Baum gehören

Die folgenden vier Knoten gehören zum Dokument, sind aber keine Baumknoten:

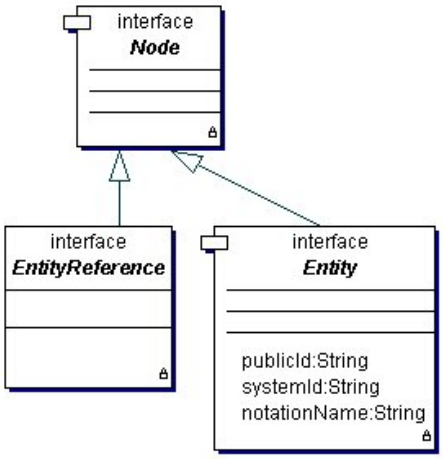
- *Attribut* Knoten
- *Entity* Knoten
- *Notation* Knoten
- *Document Fragment* Knoten

Attribut Knoten sind Element Knoten angehängt; sie sind aber nicht Child Knoten dieser Element Knoten.

Entity und Notation Knoten sind Properties des Dokument Knoten.

Document Fragment Knoten können nur im Memory aufgebaut werden.

### 9.4.5.1. Entity Knoten




*Entity* Knoten sollten nicht mit den Entity Referenz Knoten verwechselt werden!

*Entity* Knoten repräsentieren geparste und ungeparste Entities, welche in einer DTD deklariert werden.

Beim Lesen der DTD durch den Parser hängt der Parser die Entity Knoten dem Document Knoten an. Intern geschieht dies mithilfe einer indexierten Liste, einem Map (<index, entity>) Als Index wird der Entity Name verwendet. Der Index kann eingesetzt werden, um Entities mit Entity Referenzen zu verknüpfen.

Wie Sie aus dem obigen UML Diagramm erkennen, wird

1) das Entity Interface aus dem Node Interface abgeleitet (es erweitert das Node Interface)
2) das Entity Interface besitzt die Properties (private Variablen mit get/set Methoden)
   - System ID (muss)
   - Public ID (kann)
   - Name (muss)

Beim Lesen der DTD geht der Parser folgendermassen vor:

- er liest die DTD
- er list die Entity
- er ordnet dem Entity Knoten Kinder zu, welche den jeweiligen Ersatztext (der Entity) enthalten.
  Dieser Text ist Read Only!

**Beispiel 1**

```
<!ENTITY Hans "&imGlueck;">
```

Beim Lesen legt der Parser einen Entity Knoten an,

- mit dem Namen **Hans**,
- einer leeren public ID
- einer leeren System ID, sofern die Entity eine interne ist
- einem Kind Knoten, einem read-only Text Knoten, der den Text **imGlueck** enthält.

**Beispiel 2**

```
<!ENTITY Copyright SYSTEM "Copyright.xml">
```

Beim Lesen legt der Parser einen Entity Knoten an,

- mit dem Namen **Copyright**,
- einer leeren public ID
- der System ID **Copyright.xml**
- der Kind Knoten enthält den Inhalt der XML Datei und ist read-only.

#### 9.4.5.2. Notation Knoten

Zur Erinnerung: *Notations* sind "Verarbeitungshinweise" an die interpretierende Software, wenn Sie externe Daten in XML einbinden, Grafiken, Multimedia, Java-Applets oder dergleichen. Solche Daten werden vom XML-Parser nicht direkt verarbeitet. Mit Hilfe der Notations steht jedoch eine Möglichkeit zur Verfügung, der XML-verarbeitenden Software Details über die referenzierten Daten mitzuteilen.

Um externe Daten in XML-Dateien einzubinden, benötigen Sie *Entities für externe Ressourcen* und *Attribute mit Entity-Wert*. Mit einer zusätzlichen Notations-Definition geben Sie dann an, wie diese eingebundenen Daten verarbeitet werden sollen oder um welchen Datentyp es sich dabei handelt. Im letzteren Fall könnte der Parser diese Angabe z.B. ans Betriebssystem weiterreichen, das für den entsprechenden Datentyp möglicherweise ein geeignetes Programm kennt.

**Schema zur Definition von Notations**

Notations werden innerhalb einer DTD nach folgendem Schema notiert:

```
<!NOTATION Datentyp [SYSTEM|PUBLIC] "Verarbeitungshinweis" >
```

**Erläuterung**

Datentyp ist z.B. GIF. Hinter dem Datentyp ist eines der Schlüsselwörter `SYSTEM` oder `PUBLIC` erlaubt. Anschließend folgt, in Anführungszeichen gesetzt, der Verarbeitungshinweis, dessen Inhalt sich nach dem Typ (`SYSTEM` oder `PUBLIC`) richtet.

Eine solche Notation-Definition können Sie an irgendeiner Stelle innerhalb der DTD definieren - vor oder nach anderen Definitionen wie `<!ELEMENT...>`, `<!ATTLIST...>` oder `<!ENTITY...>`.

`SYSTEM`:
Auf dem lokalen Rechner oder einem Rechner im Rechnerverbund steht ein Programm für die Verarbeitung der Daten zur Verfügung steht.

**Beispiel**

```
<!NOTATION gif PUBLIC "+//ISBN 0-7923-9432-1::Graphic Notation//NOTATION CompuServer Graphic Interchange Format//EN">

<!ENTITY cs SYSTEM "Compuserver.gif" NDATA gif>
<!ELEMENT bild EMPTY>
<!ATTLIST illustration
    quelle ENTITY #REQUIRED
>
```

mit einem Beispiel XML Dokument:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE bild SYSTEM "Kap9_Beispiel_4.dtd">
<bild/>
```

In der Praxis wird die Verarbeitung nicht einheitlich gehandhabt. Das gleiche gilt für den **`PUBLIC`** Fall.

Im Rahmen von DOM werden die in der DTD vorhandene Notation durch Notation Knoten repräsentiert und dem Document Type Knoten angehängt (nicht als Child, sondern wie oben in Form einer Map: <index, inhalt> wobei der Notationsname der Index ist).

Der Notation Knoten enthält folgende Informationen:

- Name der Notation
- Public ID
  (exclusiv) oder
  System ID (je nachdem welcher bei der Definition in der DTD verwendet wurde)

**Beispiel**

```
<!NOTATION Test SYSTEM http://www.test.org/tester>
```

Analyse:

- Name : **Test**
- SYSTEM ID : ‘**http://www.test.org/tester**
- PUBLIC ID ist null

#### 9.4.5.3. Document Fragment Knoten

Ein *Document Fragment* Knoten ist sozusagen Ersatz für den Würzel Knoten eines DOM Baumes. Dieser Knoten kann alles enthalten, was ein *Element* Knoten enthalten kann:

- weitere Element Knoten
- Text Knoten
- Processing Instruction Knoten
- Comment Knoten
- ...

Document Fragment Knoten werden nicht vom Parser, sondern ausschliesslich durch Anwendungsprogramme erzeugt, beispielsweise beim Extrahieren von Teilen eines XML Dokuments (um Teile zu verschieben, kopieren, ...).

Document Fragmente sind beim Bearbeiten eines DOM Baumes ein sehr wichtiges Hilfsmittel. Gäbe es dies nicht, müssten Sie eigene Konstrukte bauen, um Teilbäume kreieren und bearbeiten zu können.

### 9.4.6. Zusammenfassung – Bauminhalte

| **Knotentyp** | **Name** | **Wert** | **Parent** | **Child(ren)** |
|---|---|---|---|---|
| Document | #document | null | null | Comment, Processing Instruction, 1 Element |
| Document Type | Name in der DOCTYPE Deklaration | null | Document | Keine |
| Element | präfixed Name | null | Element, Document, oder Document Fragment | Comment, Processing Instruction, Text, Element, Entity reference, CDATA Section |
| Text | #text | Text des Knoten | Element, Attr, Entity oder Entity Referenz | Keine |
| Attr | präfixed Name | Normalisierter Attribut Wert | Element | Text, Entity Referenz |
| Comment | #comment | text of comment | Element, Document oder Document Fragment | Keine |
| Processing Instruction | Target | Daten | Element, Document oder Document Fragment | Keine |
| Entity Reference | Name | null | Element oder Document Fragment | Comment, Processing Instruction, Text, Element, Entity Referenz, CDATA Section |
| Entity | Entity Name | null | null | Comment, Processing Instruction, Text, Element, Entity Referenz, CDATA Section |
| CDATA section | #cdata-section | Text der Section | Element, Entity oder Entity Reference | Keine |
| Notation | Notation Name | null | null | Keine |
| Document fragment | #document-fragment | null | null | Comment, Processing Instruction, Text, Element, Entity Reference, CDATA Section |

**Achtung**
Die folgenden Bestandteile eines XML Dokuments sind nicht Teile des W3C DOM Baumes:

- die XML Deklaration (Version, Standalone, Encoding)
  Diese werden ab DOM Level 3 in den Baum aufgenommen.
- Informationen aus dem DTD oder XML Schema fehlen im DOM Level 2 Baum
- White Spaces *ausserhalb* des Root Elementes werden ignoriert

DOM Parser haben aber auch noch andere Probleme:

- falls Sie ein Dokument mit einem bestimmten Encoding lesen und gleich wieder serialisieren (also in eine Datei schreiben)
  dann ist das Encoding des serialisierten Dokuments unter Umständen vom ursprünglichen Encoding verschieden.

  Der Parser hat keine Chance das Encoding zu rekonstruieren.

### *9.5. DOM Parser in Java*

Wie wir gesehen haben besteht die DOM Spezifikation aus Interfaces, nicht aus Klassen. Daher können unterschiedliche Parser DOM unterschiedlich implementieren. Die bekanntesten DOM Parser sind:

- Xerces von Apache
- Crimson von Apache und Sun
- Oracle XML Parser,
- JAXP von GNU

Jede DOM Implementierung fügt noch spezielle Erweiterungen hinzu. Daher ist es in der Regel nicht problemlos den Parser zu ändern. So wird die Validation mithilfe von XML Schemas nicht immer unterstützt.

JAXP ist ab Java SDK 1.4 standardmässig in Java vorhanden. JAXP unterstützt das Kreieren neuer XML Dokumente im Memory und die anschliessende Serialisierung als DOM Dokument.

DOM Level 3 wird in der Regel noch nicht unterstützt.

### *9.6. Parsen von XML Dokumente mit dem DOM Parser*

Der DOM Parser ist je nach Parser in einer andern Klasse:

- Xerces : **`org.apache.xerces.parsers.DOMParser`**
- Crimson : **`org.apache.crimson.jaxp.DocumentBuilderImpl`**

Auch die Methoden und Superklassen sind nicht standardisiert.
Das Lesen eines XML Dokuments in *Xerces* geschieht beispielsweise mit einer der Methoden:

- `public void` **`parse`**`(InputSource` *`source`*`)`
  `throws SAXException, IOException`
- `public void` **`parse`**`(String` *`systemID`*`) …`

Nach dem Parsen kann mit der Methode

- **`public Document getDocument()`**

das Dokument bestimmt werden.

In Crimson sieht das Ganze etwas anders aus:

- `public Document parse(InputSource source) throws SAXException, IOException`
- `public Document parse(String url) throws SAXException, IOException`
- `public Document parse(File file) throws SAXException, IOException`
- `public Document parse(InputSream in) throws SAXException, IOException`
- `public Document parse(InputSream in, String systeID) throws SAXException, IOException`

**Beispiel**

Überprüfen auf die Wohlgeformtheit eines XML Dokuments

```
/*
* WohlgeformtTest
 *
*/
package xerces;

import org.apache.xerces.parsers.DOMParser;
import org.xml.sax.SAXException;
import java.io.IOException;

public class WohlgeformtTest {
      private static final String URL="adressbuch_01.xml";
      public static void main(String[] args) {
            String document;

            if (args.length <= 0) {
                  System.out.println("Usage: java WohlgeformtTest URL");
                  //return;
                  document = URL;
            } else {
                  document = args[0];
            }

            DOMParser parser = new DOMParser();
            try {
                  parser.parse(document);
                  System.out.println(document + " ist wohlgeformt.");
            } catch (SAXException e) {
                  System.out.println(document + " ist nicht wohlgeformt.");
            } catch (IOException e) {
                  System.out.println(
                        "Fehler beim Lesen von "+document);
            }
      }
}
```

Das Programm versucht die XML Datei zu lesen. Falls diese nicht gelingt, wird eine IOException geworfen. Gelingt dies, dann wird im Falle eines Fehlers ausgegeben, dass das Dokument nicht wohlgeformt ist.

Nun schauen wir uns das selbe Programm mit dem Crimson Parser realisiert an. Die Funktionsweise ist die Gleiche wie oben. Wesentlich anders ist der Aufbau des Programm:

- Crimson baut zuerst eine Document Builder Factory
- Dann wird das Dokument geparst / eingelesen.

```
/*
```

```
* WohlgeformtTest
 *
*/
package crimson;

import java.io.File;
import java.io.IOException;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.xml.sax.SAXException;

public class WohlgeformtTest {
	private static final String URL = "adressbuch_01.xml";
	public static void main(String[] args) {
		String document;

		if (args.length <= 0) {
			System.out.println("Usage: java WohlgeformtTest URL");
			//return;
			document = URL;
		} else {
			document = args[0];
		}

		// DOM
		try {
			DocumentBuilderFactory factory
			 = DocumentBuilderFactory.newInstance();
			DocumentBuilder parser = factory.newDocumentBuilder();
			parser.parse(document);
			// Document
			Document doc = parser.parse(new File(document));
			System.out.println(document + " ist wohlgeformt.");
		} catch (SAXException e) {
			System.out.println(document + " ist nicht wohlgeformt.");
		} catch (IOException e) {
			System.out.println("Fehler beim Lesen von " + document);
		} catch (ParserConfigurationException e) {

			e.printStackTrace();
		} catch (FactoryConfigurationError e) {

			e.printStackTrace();
		}
	}
}
```

### 9.6.1. wie wählt JAXP den Parser aus?

JAXP ist im Wesentlichen Parser unabhängig, arbeit mit Factories und es ist kaum erkennbar, welchen Parser JAXP konkret einsetzt oder ob der Parser im Hintergrund ausgetauscht werden könnte, ohne Programmänderungen.

Sie können den Parser mithilfe der **`javax.xml.parsers.DocumentBuilderFactory`** System Eigenschaft setzen

**Variante 1:**
Auf der Kommandozeile

```
java – Djava.xml.parsers.DocumentBuilderFactory =
org.apache.xerces.jaxp.DocumentBuilderFactotyImpl ...
```

**Variante 2:**
Im **`JRE\lib`** Verzeichnis kann man eine Datei **`jaxp.properties`** anlegen und den Parser spezifizieren:

```
javax.xml.parsers.DocumentBuilderFactory = xyz.xml.dom.JAXPFactory
```

**Variante 3:**
Wenn all das versagt: JAXP sucht gezielt eine konkrete Unterklasse der **`DocumentBuilderFactory`** Klasse in **`META-INF/services/javax.xml.parsers.DocumentBuilderFactory`** allen JAR Dateien.

### 9.6.2. Konfiguration der DocumentBuilderFactory

Der Parser kann mit einigen Optionen gebaut werden. Diese müsen je nach Anwendung gesetzt / enabled oder sogar in eigenen Klassen implementiert werden.

#### 9.6.2.1. Coalescing

Wenn Sie die CDATA Sections mit den Text Knoten mischen wollen, muss diese Eigenschaft auf **`false`** stehen.

```
public boolean isCoalescing();
public void setCoalescing(boolean coalescing)
```

Standardwert ist false. In der Regel sollten Sie diese Eigenschaften aber auf true setzen, beispielsweise falls Sie einfach hein XML Dokument lessen und eventuell wieder schreiben wollen:

- in diesem Fall sollten CDATA Sections wie normaler Text behandelt werden

#### 9.6.2.2. Auflösen von Entity Referenzen

Die Eigenschaft, ob eine Parser Entitäten auflöst oder nicht, kann mit folgenden Methoden abgefragt bzw. gesetzt werden:

- `public boolean isExpandEntityReferences();`
- `public void setExpandEntityReferences(boolean expandEntityReferences);`

Standardwert ist **`true`**.
Falls ein Parser validierend ist, dann wird er auch die Entity Referenzen auflösen, selbst wenn dieses Feature ausgeschaltet ist.

#### 9.6.2.3. Kommentare ignorieren

Wenn Sie Kommentare ignorieren wollen, können Sie dies mithilfe der folgenden Methoden erreichen bzw. abfragen:

- `public boolean isIgnoringComments();`
- `public void setIgnoringComments(boolean ignoringComments);`

Standardwert ist **`false`**, Kommentare werden also nicht ignoriert.

#### 9.6.2.4. White Spaces in Element Knoten ignorieren

Ob White Spaces ignoriert oder als Text interpretiert wird, hängt von der folgenden Einstellung ab:

- `public boolean isIgnoringElementContentWhitespace();`
- `public void setIgnoringContentWhitespace(boolean ignore);`

Standardwert ist **`false`**. Damit werden die White Spaces als Text Knoten berücksichtigt. Falls Sie die Eigenschaft einschalten geschieht nichts, ausser Sie haben dem Dokument eine DTD zugeordnet und das Dokument ist ein gültiges Dokument. In allen andern Fällen sagt der Parser zwar, dass White Spaces ignoriert werden; sie werden es aber nicht!

#### 9.6.2.5. Einsatz von Namespaces

Die Grundidee hinter XML Namensräumen ist:

- generelle Lösung zur Behebung von Mehrdeutigkeiten in Elementnamen und Attributnamen aus verschiedenen Kontexten
  unter
- Beibehaltung möglichst grosser Freiheit bei der Erstellung eigener Vokabulare (als Sammlungen von zusammengehörigen Begriffen / Tags)

Namensräume beziehen sich somit auf Elemente und Attribute.

**Beispiel**

In der einen Applikation wird der Kunde als die zentrale Grösse angesehen. Im unterstellt ist die Kundenadresse, die Lieferadresse und viele weitere Informationen.

In der andern Applikation ist der Kunde ein Spezialfall eines Business Partners. Der Business Partner besitzt Adressen, nicht der Kunde.

Die beiden Informations-Hierarchien sehen somit völlig unterschiedlich aus. Es ist auch nicht klar, von welcher Kunden sicht gesprochen wird, falls ein **`<Kunde>...</Kunde>`** Tag auftaucht.

**Lösungsvarianten 1**

Diese Mehrdeutigkeit könnte man beheben, indem wir einen komplexeren Tagnamen verwenden:

```
<BusinessPartner.Kunde.Adresse>...</BusinessPartner.Kunde.Adresse >
```

bzw.

```
<Kunde.LieferAdresse>...</ Kunde.LieferAdresse >
```

**Lösungsvarianten 2**

Branchenspezifische fixe Definition der Tags. Dieser Weg wurde im EDI /EDIFACT Umfeld gewählt und in XML Umfeld kaum generell machbar.

Gemäss W3C sieht die Lösung folgendermassen aus:

1) Bindung einer URI an ein frei wählbares Präfix durch ein standardisiertes Attribut
   Beispiel:
   **<rechnung xmlns:rech="http://www.inkasso.com">**
   (xmlns steht für XML NameSpace und ist „reservierter Name“)
2) Qualifizieren der Element- und Attribut-Namen mit dem gerade definierten Präfix.
   Beispiel:

```
<rech:rechnung xmlns:rch="http://www.inkasso.com">
      <rech:kunde>
            <rech:kundenNr>1234</rech:kundenNr>
      </rech:kunde>
...
```

   Oder
3) Definition eines Namensraumes ohne Präfix
   **<rechnung xmlns="http://www.inkasso.com">**
   Dann sind alle untergeordneten Tags implizit in diesem Namensraum:

```
<rechnung:rechnung xmlns ="http://www.inkasso.com">
      <!—diese Tags haben implizit den Präfix‚rechnung‘ //-->
      <kunde>
            <kundenNr>1234</kundenNr>
      </kunde>
```

   **"rechnung"** ist der „Default Namespace“ für diese Tags und untergeordnete Tags.

Dabei wird als eindeutiger Name eine URI verwendet, also keine reale URL!
In Java stehen zwei Methoden zur Verfügung:

- `public boolean isNamespaceAware();`
- `public void setNamespaceAware(boolean namespaceAware);`

Standardwert ist **false**; aber Sie sollten diesen Standardwert umgehend auf **true** setzen.

#### 9.6.2.6. Validierend

Die Validierung eines Dokuments bedingt, dass Sie eine DTD im XML Dokument angeben.

Das Setzen oder Abfragen des Tags ist einfach:

- `public boolean isValidating();`
- `public void setValidating(boolean validating);`

Standardwert der Eigenschaft ist **false**.

Wenn Sie validieren wollen, müssen Sie auch noch einen **ErrorHandler** kreieren und beim **DocumentBuilder** registrieren.

**Beispiel**
Das Dokument wird mit einer DTD validiert. Die XML Datei ist gültig, extern mit XML Spy überprüft.
Die DTD ist korrekt.

```
/*
* ValidierenderParser
 *
*/
```

```
package crimson;

import javax.xml.parsers.*; // JAXP
import org.xml.sax.*;
import java.io.IOException;

public class ValidierenderParser {
      private static final String URL="adressbuch_02.xml";
      public static void main(String[] args) {
            String document;
            if (args.length <= 0) {
                  System.out.println(
                        "Usage: java ValidierenderParser URL");
                  //return;
                  document = URL;
            } else document = args[0];

            try {
                  DocumentBuilderFactory factory =
                        DocumentBuilderFactory.newInstance();
                  //Namespace
                  factory.setNamespaceAware(true);
                  //Validation
                  factory.setValidating(true);

                  DocumentBuilder parser = factory.newDocumentBuilder();

                  //Error Handler
                  ErrorHandlerBeispiel handler =
                                          new ErrorHandlerBeispiel();
                  parser.setErrorHandler(handler);

                  parser.parse(document);
                  if (handler.isValid()) {
                        System.out.println(document + " ist gültig.");
                  } else {
                        // falls nicht wohlgeformt:
                        // Exception wurde bereits geworfen
                        System.out.println(document + " ist wohlgeformt.");
                  }

            } catch (SAXException e) {
                  System.out.println(document + " ist nicht wohlgeformt.");
            } catch (IOException e) {
                  System.out.println(
                        "IOException:
                              es wurde keine Überprüfung durchgeführt "
                              + document);
            } catch (FactoryConfigurationError e) {
                  System.out.println("Die Factory Klasse wurde nicht
                                                            gefunden");
            } catch (ParserConfigurationException e) {
                  System.out.println("JAXP Parser wurde nicht gefunden");
            }
      }
}
```

Als Ausgabe erhalten wir:

```
Usage: java ValidierenderParser URL
adressbuch_02.xml ist gültig.
```

Jetzt ändern wir die XML Datei und fügen irgendwo einen beliebigen Tag ein:
Als Ausgabe ergibt sich:

```
Error: Element type "Hilfe" must be declared.
 in Zeile 15, Spalte 10
Error: The content of element type "Eintrag" must match "(Person?,Firma?)".
 in Zeile 22, Spalte 12
adressbuch_02.xml ist wohlgeformt.
```

Die wilde Mischung der Fehlermeldungssprachen vergessen wir mal.

#### 9.6.2.7. Parser spezifische Attribute

Je nach Parser, auch bei den JAXP konformen, können weitere Features möglich sein.

**Beispiel**
In Xerces können Sie verlangen, dass Entity Referenz Knoten in den Baum aufgenommen werden. In diesem Fall wird der Entity Text als Child Text Knoten angehängt.

JAXP sieht solche Erweiterungen bereits vor und hat einen generischen Mechanismus eingebaut:

```
- public Object getAttribute(String name)
            throws IllegalArgumentException;
- public void setAttribute(String name, Object value)
            throws IllegalArgumentException;
```

### 9.6.3. DOM Level 3

JAXP ist für DOM Level 2 der gängige Parser. W3C bemüht sich, die Ideen aus JAXP zu verallgemeinern und als Standard in DOM Level 3 einzubauen.

DOM Level 3 ist aber noch im Fluss:

- http://www.w3.org/TR/2004/PR-DOM-Level-3-Core-20040205 ist zur Zeit eine Proposed Recommendation
- bis Parser DOM Level 3 unterstützen vergeht sicher noch einige Zeit.

Gemäss Xerces:
„The DOM Level 3 specification is at the stage of **Proposed Recommendation**, which represents work in progress and has been widely reviewed and satisfies certain technical requirements but may still change based on implementation experience. This implementation is experimental: it should not be considered as complete or correct.
“

### 9.7. Das Node Interface

Das gute am DOM Modell (gemäss W3C) ist, dass, nachdem Sie erst einmal einen Parsebaum (Document) gebildet haben, dieser vom konkreten Parser unabhängig ist. Sie können danach mit den Standard DOM Interfaces weiterarbeiten.

Zentrales Interface ist, wie wir schon gesehen haben, das **Node** Interface:

```
package org.w3c.dom;
public interface Node {
    // NodeType
    public static final short ELEMENT_NODE              = 1;
    public static final short ATTRIBUTE_NODE            = 2;
    public static final short TEXT_NODE                 = 3;
    public static final short CDATA_SECTION_NODE        = 4;
    public static final short ENTITY_REFERENCE_NODE     = 5;
    public static final short ENTITY_NODE               = 6;
    public static final short PROCESSING_INSTRUCTION_NODE = 7;
    public static final short COMMENT_NODE              = 8;
    public static final short DOCUMENT_NODE             = 9;
    public static final short DOCUMENT_TYPE_NODE        = 10;
    public static final short DOCUMENT_FRAGMENT_NODE    = 11;
    public static final short NOTATION_NODE             = 12;
/*    Node Eigenschaften*/
    public String getNodeName();
    public String getNodeValue() throws DOMException;
    public void setNodeValue(String nodeValue) throws DOMException;
    public short getNodeType();
    public String getNamespaceURI();
    public String getPrefix();
    public void setPrefix(String prefix) throws DOMException;
    public String getLocalName();
/*    Navigationsmethoden*/
    public Node getParentNode();
    public boolean hasChildNodes();
    public NodeList getChildNodes();
    public Node getFirstChild();
    public Node getLastChild();
    public Node getPreviousSibling();
    public Node getNextSibling();
    public boolean hasAttributes();
    public NamedNodeMap getAttributes();
    public Document getOwnerDocument();
/*    Manipulationsmethoden*/
    public Node insertBefore(Node newChild, Node refChild)
                                        throws DOMException;
    public Node replaceChild(Node newChild, Node oldChild)
                                        throws DOMException;
    public Node removeChild(Node oldChild) throws DOMException;
    public Node appendChild(Node newChild) throws DOMException;
/*    Hilfsmethoden*/
    public Node cloneNode(boolean deep);
    public void normalize();
    public boolean isSupported(String feature,
                               String version);
}
```

Nun gehen wir auf die einzelnen oben ausgezeichneten Bereiche vertieft ein.

### 9.7.1. Knoten Typen

Jedem Knoten Typ ist eine Konstante zugeordnet. Zudem steht eine Methode zur Verfügung, welche als Rückgabewert eine dieser Konstanten liefert. Diese Konstanten sind vom Java Basisdatentyp **`short`**:

```
    public short getNodeType();
```

Was nicht geht, ist die Abfrage des Klassentyps eines Knotens (mit **`instanceof.getClass()`** ).

DOM wurde definitiv nicht für Java geschrieben. Die Spezifikation wurde in IDL (OMG.org) erstellt, und IDL lehnte sich (aus historischen Gründen) eher an C/C++ an.

**Beispiel**
Die folgende Klasse kann benutzt werden, um die interne Codierung der Knoten Typen in normalen Text umzusetzen.

```
package crimson;
import org.w3c.dom.Node;

public class NodeInterfaceText {
      public static String getTypeName(Node node) {
            int knotenTyp = node.getNodeType();
            /* upcast short auf int */
            switch (knotenTyp) {
                  case Node.ELEMENT_NODE :
                        return "Element";
                  case Node.ATTRIBUTE_NODE :
                        return "Attribute";
                  case Node.TEXT_NODE :
                        return "Text";
                  case Node.CDATA_SECTION_NODE :
                        return "CDATA Section";
                  case Node.ENTITY_REFERENCE_NODE :
                        return "Entity Reference";
                  case Node.ENTITY_NODE :
                        return "Entity";
                  case Node.PROCESSING_INSTRUCTION_NODE :
                        return "Processing Instruction";
                  case Node.COMMENT_NODE :
                        return "Comment";
                  case Node.DOCUMENT_NODE :
                        return "Document";
                  case Node.DOCUMENT_TYPE_NODE :
                        return "Document Type Declaration";
                  case Node.DOCUMENT_FRAGMENT_NODE :
                        return "Document Fragment";
                  case Node.NOTATION_NODE :
                        return "Notation";
                  default :
                        return "unbekannter Knotentyp";
            }
      }
}
```

### 9.7.2. Knoten Eigenschaften

Als nächstes wollen wir die Eigenschaften der Knoten bestimmen (und ausgeben). Wie eine entsprechende Klasse aussehen könnte zeigt das folgende Beispiel.

**Beispiel**

```
package crimson;
import org.w3c.dom.*;
import java.io.*;

public class NodePropertiesText {
  private Writer out;
  public NodePropertiesText(Writer out) {
      if (out == null) {
        throw new NullPointerException("Writer ist null.");
      }
      this.out = out;
  }
  public NodePropertiesText() {
      this(new OutputStreamWriter(System.out));
  }
  private int nodeCount = 0;
  public void writeNode(Node node) throws IOException {
      if (node == null) {
        throw new NullPointerException("Node ist null.");
      }
      if (node.getNodeType() == Node.DOCUMENT_NODE
       || node.getNodeType() == Node.DOCUMENT_FRAGMENT_NODE) {
        // zurücksetzen des Zählers (neues Dokument)
        nodeCount = 1;
      }
      String name      = node.getNodeName();
      String type      = NodeInterfaceText.getTypeName(node);
      String localName = node.getLocalName();
      String uri       = node.getNamespaceURI();
      String prefix    = node.getPrefix();
      String value     = node.getNodeValue();
      StringBuffer result = new StringBuffer();
      result.append("Node " + nodeCount + ":\r\n");
      result.append("  Type: " + type + "\r\n");
      result.append("  Name: " + name + "\r\n");
      if (localName != null) {
        result.append("  Local Name: " + localName + "\r\n");
      }
      if (prefix != null) {
        result.append("  Prefix: " + prefix + "\r\n");
      }
      if (uri != null) {
        result.append("  Namespace URI: " + uri + "\r\n");
      }
      if (value != null) {
        result.append("  Value: " + value + "\r\n");
      }
      out.write(result.toString());
      out.write("\r\n");
      out.flush();
      nodeCount++;
  }
}
```

Die Ausgabe ist zu lange für das Adressbuch. Hier der Start der Ausgabe:

```
Node 1:
  Type: Document
  Name: #document

Node 2:
  Type: Comment
  Name: #comment
  Value:  DTD für das Dokument. Um zu aktivieren, Kommentarzeichen
entfernen

Node 3:
  Type: Document Type Declaration
  Name: Adressbuch

Node 4:
  Type: Element
  Name: Adressbuch
  Local Name: Adressbuch

Node 5:
  Type: Text
  Name: #text
  Value:


Node 6:
  Type: Comment
  Name: #comment
  Value:  Das Adressbuch kann Einträge zu Personen oder Firmen enthalten
```

zum XML Dokument:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- DTD für das Dokument. Um zu aktivieren, Kommentarzeichen entfernen -->
<!DOCTYPE Adressbuch SYSTEM "adressbuch_02.dtd">
<Adressbuch>
      <!-- Das Adressbuch kann Einträge zu Personen oder Firmen enthalten -
->
      <Eintrag>
...
      </Eintrag>
```

Die Methoden sind mehrfach überladen, um die unterschiedlichen Knoten Typen ausgeben zu können.

### 9.7.3. Navigieren im Baum

Nun wollen wir uns durch den Baum bewegen und beispielsweise spezielle Knoten suchen und finden.

Die erste Frage: besitzt der Knoten überhaupt Child Knoten?
Lösung: die Methode **`hasChildren()`** liefert einen **`boolean`**

Die Abfrage der Kindknoten geschieht dann mithilfe einer der Methoden

- **`getFirstChild()`**
- **`getLastChild()`**

Unter Umständen besitzt der Knoten auch noch Attribute. Diese müssen wir speziell abfragen und bestimmen:

- **`hasAttributes()`**
- **`getAttributes()`**

Da der DOM ein Baum ist, kann man ihn am besten mit rekursiven Methoden manipulieren und traversieren.

```
package crimson;
import java.io.IOException;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Node;
import org.xml.sax.SAXException;

public class NodeInterfaceTextTest {
      private static final String URL = "adressbuch_02.xml";
      public static void main(String[] args) {
            String doc;
            if (args.length <= 0) {
            System.out.println("Usage: java NodeInterfaceTextTest URL");
                  //return;
                  doc = URL;
            } else {
                  doc = args[0];
            }

            NodeInterfaceTextTest iterator = new NodeInterfaceTextTest();
            try {
                  //JAXP sucht den Parser
                  DocumentBuilderFactory factory =
                        DocumentBuilderFactory.newInstance();
                  // Namespace Support
                  factory.setNamespaceAware(true);
                  DocumentBuilder parser = factory.newDocumentBuilder();

                  // einlesen des Dokuments
                  Node document = parser.parse(doc);

                  // DOM Baum analysieren, ab dem Root
                  iterator.followNode(document);
```

```
            } catch (SAXException e) {
                  System.out.println(doc + " ist nicht wohlgeformt.");
                  System.out.println(e.getMessage());
            } catch (IOException e) {
                  System.out.println(e);
            } catch (ParserConfigurationException e) {
                  System.out.println("JAXP Parser wurde nicht gefunden");
            }
      }

      private NodePropertiesText printer = new NodePropertiesText();
      // rekursive Traversierung

      public void followNode(Node node) throws IOException {
            printer.writeNode(node);
            if (node.hasChildNodes()) {
                  Node firstChild = node.getFirstChild();
                  followNode(firstChild);
            }
            Node nextNode = node.getNextSibling();
            if (nextNode != null)
                  followNode(nextNode);
      }
}
```

Die Ausgabe sieht wie vorne aus:

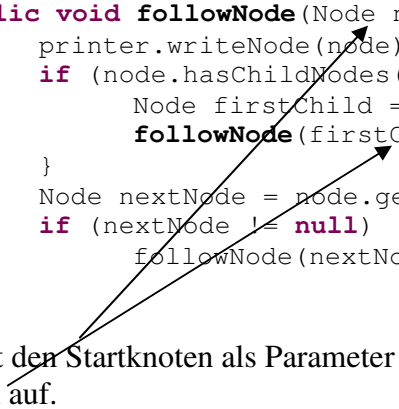
```
Node 66:
  Type: Element
  Name: Homepage
  Local Name: Homepage

Node 67:
  Type: Text
  Name: #text
  Value: www.arvenflores.ch

Node 68:
  Type: Text
  Name: #text
  Value:


Node 69:
  Type: Text
  Name: #text
  Value:


Node 70:
  Type: Text
  Name: #text
  Value:
```

bei folgendem XML

```
                  <Homepage>www.arvenflores.ch</Homepage>
            </Firma>
      </Eintrag>
</Adressbuch>
```

Der Schlüssel zum Verständnis des Programms ist die rekursive Methode

```
public void followNode(Node node) throws IOException {
      printer.writeNode(node);
      if (node.hasChildNodes()) {
            Node firstChild = node.getFirstChild();
            followNode(firstChild);
      }
      Node nextNode = node.getNextSibling();
      if (nextNode != null)
            followNode(nextNode);
}
```



Diese erhält den Startknoten als Parameter und ruft dann sich selbst mit dem ersten Kindknoten auf.

### 9.7.4. DOM Bäume modifizieren

Falls Sie einen DOM Baum von Grund auf im Memory aufbauen wollen, benötigen Sie die Methoden

- `public Node insertBefore(Node neu, Node vorgänger) throws...`
- `public Node replaceChild(Node, neu, Node alt) throws ...`
- `public Node removeChild(Node knoten) throws DOMException`
- `public Node appendChild(Node append) throws DOMException`

Die zwei mittleren Methoden schneiden Knoten aus einem bestehenden Baum aus, ohne allerdings den Bezug zum DOM Baum zu verlieren.

Entfernte Knoten können auch in einen andern DOM wieder eingefügt werden (Knotentausch).

Die folgende Klasse verschiebt Processing Instructions und Kommentare nach oben (an den Anfang des XML Dokuments).

```
package crimson;

import org.w3c.dom.DOMException;
import org.w3c.dom.Node;

public class KnotenUmordnen {
      public void processNode(Node current) throws DOMException {

            Node nextSibling = current.getNextSibling();

            int nodeType = current.getNodeType();
            if (nodeType == Node.COMMENT_NODE
                  || nodeType == Node.PROCESSING_INSTRUCTION_NODE) {

                  Node document = current.getOwnerDocument();
                  Node root = document.getFirstChild();
                  while (!(root.getNodeType() == Node.ELEMENT_NODE)) {
                        root = root.getNextSibling();
                  }

                  Node parent = current.getParentNode();
                  parent.removeChild(current);
                  if (nodeType == Node.COMMENT_NODE) {
                        document.appendChild(current);
            } else if (nodeType == Node.PROCESSING_INSTRUCTION_NODE) {
```

```
                        document.insertBefore(current, root);
                    }
                } else if (current.hasChildNodes()) {
                    Node firstChild = current.getFirstChild();
                    processNode(firstChild);
                }

                if (nextSibling != null) {
                    processNode(nextSibling);
                }
        }
}
```

Hinweis:
Das Programm hat noch einen Fehler. Ich wird ihn umgehend korrigieren.

### 9.7.5. Hilfsmethoden

Die folgenden drei Methoden erfüllen Hilfsfunktionen:

- `public Node cloneNode(Boolean deep);`
- `public void normalize();`
- `public void isSupported(String feature, String version);`

#### 9.7.5.1. normalize()

Die Methode fasst mehrere Text Knoten unterhalb dieses Knotens auf eine normalisierte Art und Weise zusammen, auch Attribut Knoten.

In dieser normalisierten Form trennen nur Strukturknoten (Elemente, Kommentare, Processing Instruction Knoten, CDATA Sections und Entity Referenzen) die Text Knoten.

Mit andern Worten:

- es gibt keine benachbarten Text Knoten mehr
- es gibt keine leeren Text Knoten mehr.

Die Methode kann direkt nach der Konstruktiondes Parsebaumes ausgeführt werden:

- **`Document doc = parser.parse(xml_doc);`**
- **`Document.normalize();`**

#### 9.7.5.2. cloneNode()

Die Methode

- `public Node cloneNode(Boolean deep)`

Die Methode

- `public Node cloneNode(Boolean deep);`

liefert eine Kopie dieses (aktuellen) Knotens. Allerdings besitzt die Kopie keinen Eltern Knoten (**`parentNode`** ist **`null`**), obschon der Knoten zum selben Dokument gehört Sie können ihn aber mit einer der Methoden **`insertbefore() appendNode(), replaceNode()`** in den Parsebaum einfügen. Beim Klonen eines Element Knotens werden alle Attribute und deren Werte, auch die vom XML Prozessor generierte, kopiert.

Die Methode kopiert Texte nur genau dann, falls eine tiefe Kopie (**`deep=true`**) gemacht wird, weil der eigentliche Text in einem Text Kind Knoten steckt.

Bei Attribut Knoten geschieht auch etwas Spezielles:

- bei geklonte Attribut Knoten ist das **specified** Flag **true**.

**Hinweis**
Falls der geklonte Knoten in einen andern Baum eingefügt werden soll, leistet die **importNode**() Methode einen besseren Dienst.

#### 9.7.5.3. isSupported()

Diese Hilfsmethode haben wir ganz am Anfang schon gestestet. Sie liefert Hinweise darauf, welche Features unterstützt werden. Die Version ist identisch mit dem DOM Level, also normalerweise 2.0.

## 9.8. Das NodeList Interface

Die Kind Knoten eines DOM Knotens werden in einem NodeList Objekt abgespeichert.
(Original Source Code)

```
package org.w3c.dom;

/**
 * The <code>NodeList</code> interface provides the abstraction of an ordered
 * collection of nodes, without defining or constraining how this collection
 * is implemented. <code>NodeList</code> objects in the DOM are live.
 * <p>The items in the <code>NodeList</code> are accessible via an integral
 * index, starting from 0.
 * <p>See also the <a href='http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113'>Document
Object Model (DOM) Level 2 Core Specification</a>.
 */
public interface NodeList {
    /**
     * Returns the <code>index</code>th item in the collection. If
     * <code>index</code> is greater than or equal to the number of nodes in
     * the list, this returns <code>null</code>.
     * @param index Index into the collection.
     * @return The node at the <code>index</code>th position in the
     *   <code>NodeList</code>, or <code>null</code> if that is not a valid
     *   index.
     */
     public Node item(int index);

    /**
     * The number of nodes in the list. The range of valid child node indices
     * is 0 to <code>length-1</code> inclusive.
     */
     public int getLength();
}
```

Eine solche Liste wird beispielsweise als Rückgabewert der Methode

- **getChildNodes()**

geliefert. Die Liste kann beispielsweise in ein Array abgefüllt werden.

**Anwendungsbeispiel**

```
public void followNode2(Node node) throws IOException {
      printer.writeNode(node);
      // Kind Knoten verarbeiten
      NodeList children = node.getChildNodes();
      for (int i = 0; i < children.getLength(); i++) {
            Node child = children.item(i);
            followNode(child); // Rekursion
      }
}
```

Die NodeList Datenstruktur ist dynamisch, verwendet “live links” auf die Daten:

- wenn Sie einen Knoten hinzufügen oder entfernen,
  dann ändert sich die Node Liste automatisch.
- Allerdings sind diese Listen (wie das meiste in DOM) nicht Thread-safe.

### 9.9. JAXP Serialisierung

Im Memory Manipulationen am DOM Baum sind ziemlich standardisiert. Ganz anders sieht die Serialisierung und Deserialisierung (lesen und schreiben in / aus Dateien) aus.

Diese Funktionen fehlen auch in JAXP. Eine Variante ist der Einsatz der Transformer Factory aus dem **`javax.xml.transform`** Package.

Hier ein systematisches Vorgehen für den Einsatz dieser Factory:

1. mithilfe der statischen Methode **`TransformerFactory.newInstance()`** wird ein neues **`javax.xml.transform.TransformerFactory`** Objekt kreiert.
2. Die Methode **`newTransformer()`** der **`TransformerFactory`** liefert eine Implementations-abhängige Instanz der abstrakten Klasse **`javax.xml.transform.Transformer`**.
3. Nun müssen wir aus dem DOM **`Document`** Objekt ein **`javax.xml.transform.dom.DOMSource`** Objekt konstruieren.
4. Dieses können wir dann mit einem neu konstruierten **`javax.xml.transform.stream.StreamResult`** Objekt verknüpfen.
5. Das **`StreamResult`** Objekt können wir nun mit einem **`OutputStream`** verknüpfen, über den wir das externe XML Dokument generieren wollen.
6. jetzt haben wir alle Objekte zusammen, um das Source und das Target Objekt an die **`transform()`** Methode des Transform Objekts aus Schritt 2 zu übergeben.

Das folgende Programm zeigt, wie dies konkret aussehen könnte:

```
import java.io.IOException;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.transform.Result;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerConfigurationException;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;

import org.w3c.dom.Document;
import org.xml.sax.SAXException;

public class TransformerTest {
      private static String URL="adressbuch_02.xml";
      public static void main(String[] args) {
            String xml_doc;
            if (args.length <= 0) {
                  System.out.println("Usage: java TransformerTest URL");
                  //return;
                  xml_doc = URL;
            } else {
```

```
                xml_doc = args[0];
            }


            try {
                // Parser bestimmen
                DocumentBuilderFactory factory =
                      DocumentBuilderFactory.newInstance();
                factory.setNamespaceAware(true);
                DocumentBuilder parser = factory.newDocumentBuilder();

                // Lesen des Dokuments
                Document document = parser.parse(xml_doc);

                // Modifizieren des Dokuments
                KnotenUmordnen.processNode(document);

                // Ausgabe
                TransformerFactory xformFactory =
                                          TransformerFactory.newInstance();
                Transformer idTransform = xformFactory.newTransformer();
                Source input = new DOMSource(document);
                Result output = new StreamResult(System.out);
                idTransform.transform(input, output);

            } catch (SAXException e) {
                System.out.println(xml_doc + " ist wohlgeformt.");
            } catch (IOException e) {
                System.out.println(
                      "IOException beim Lesen von " + xml_doc);
            } catch (FactoryConfigurationError e) {
                System.out.println("Factory Klasse nicht gefunden");
            } catch (ParserConfigurationException e) {
                System.out.println("JAXP Parser wurde nicht gefunden");
            } catch (TransformerConfigurationException e) {
                System.out.println("DOM Parser kennt 'transform' nicht.");
            } catch (TransformerException e) {
                System.out.println("Transform schlug fehl.");
            }
        }
}
```

### 9.10. DOM Exceptions

Die Exceptions im XML Verarbeiten mit Java sind eher mühsam und unpräzise.

```
package org.w3c.dom;

public class DOMException extends RuntimeException {
    public DOMException(short code, String message) {
       super(message);
       this.code = code;
    }
    public short   code;
    // ExceptionCode
    /* If index or size is negative, or greater than the allowed value*/
    public static final short INDEX_SIZE_ERR            = 1;
    /* If the specified range of text does not fit into a DOMString  */
    public static final short DOMSTRING_SIZE_ERR        = 2;
    /* If any node is inserted somewhere it doesn't belong */
    public static final short HIERARCHY_REQUEST_ERR     = 3;
    /*If a node is used in a different document than the one that created
     * (that doesn't support it) */
    public static final short WRONG_DOCUMENT_ERR        = 4;
    /*If an invalid or illegal character is specified, such as in a name.*/
    public static final short INVALID_CHARACTER_ERR     = 5;
    /*If data is specified for a node which does not support data  */
    public static final short NO_DATA_ALLOWED_ERR       = 6;
    /* If an attempt is made to modify an object where modifications are
       not allowed */
    public static final short NO_MODIFICATION_ALLOWED_ERR = 7;
    /*If an attempt is made to reference a node in a context where it does
     * not exist*/
    public static final short NOT_FOUND_ERR             = 8;
    /*If the implementation does not support the requested type of object
      or operation. */
    public static final short NOT_SUPPORTED_ERR         = 9;
    /*If an attempt is made to add an attribute that is already in use
      elsewhere*/
    public static final short INUSE_ATTRIBUTE_ERR       = 10;
    /*If an attempt is made to use an object that is not, or is no longer,
     * usable. @since DOM Level 2 */
    public static final short INVALID_STATE_ERR         = 11;
    /*If an invalid or illegal string is specified. @since DOM Level 2*/
    public static final short SYNTAX_ERR                = 12;
    /*If an attempt is made to modify the type of the underlying object.
     * @since DOM Level 2*/
    public static final short INVALID_MODIFICATION_ERR  = 13;
    /*If an attempt is made to create or change an object in a way which is
     * incorrect with regard to namespaces.* @since DOM Level 2 */
    public static final short NAMESPACE_ERR             = 14;
    /*If a parameter or an operation is not supported by the underlying
     * object.@since DOM Level 2 */
    public static final short INVALID_ACCESS_ERR        = 15;
}
```

Die DOMException ist die einzige Exception, welche von W3C gefordert wird.

### 9.11. DOM oder SAX?

Die Auswahl ist denkbar einfach:

- wenn Sie viel Memory haben und das Dokument im Memory manipulieren müssen
  dann werden Sie DOM einsetzen, trotz des viel komplexeren API's
- wenn Sie schnell etwas in einem XML Dokument heraus holen wollen,
  dann ist SAX die beste Wahl.
- Wenn Sie DOM Funktionalität benötigen, aber einfache Aufgaben erledigen wollen
  dann könnte JDOM eine gute Wahl sein.

### 9.12. Zusammenfassung

Das Document Object Model gemäss W3C bietet ein standardisiertes API für die verarbeitung von XML Dokumenten. XML Dokumente werden im Memory als Bäume dargestellt. Fast alles wird aus dem Node Interface abgeleitet.

Was bei DOM Level 2 fehlt sind klare Interfaces für das Serialisieren und Deserialisieren (lesen und schreiben von XML Dokumenten in einer standardisierten Art und Weise).