

# Programmieren II

## XML



Heusch --  
Ratz --

Institut für Automation und angewandte Informatik

```
final List<String> allResults = new ArrayList<String>();  
final Map<String, Integer> typeWordResultCount = new HashMap<String, Integer>();  
final Map<String, Integer> typePoints = new HashMap<String, Integer>();  
evaluation.put(type, typePoints);  
  
for (final Sheet sheet : this.sheets) {  
    final String sheetResult = sheet.getPlayerInput(type);  
    if (sheetResult.startsWith(start) && this.isValidWord(sheetResult, type))  
        validWordCountForType++;  
    allResults.add(sheetResult);  
}
```

## ■ XML (Extensible Markup Language)

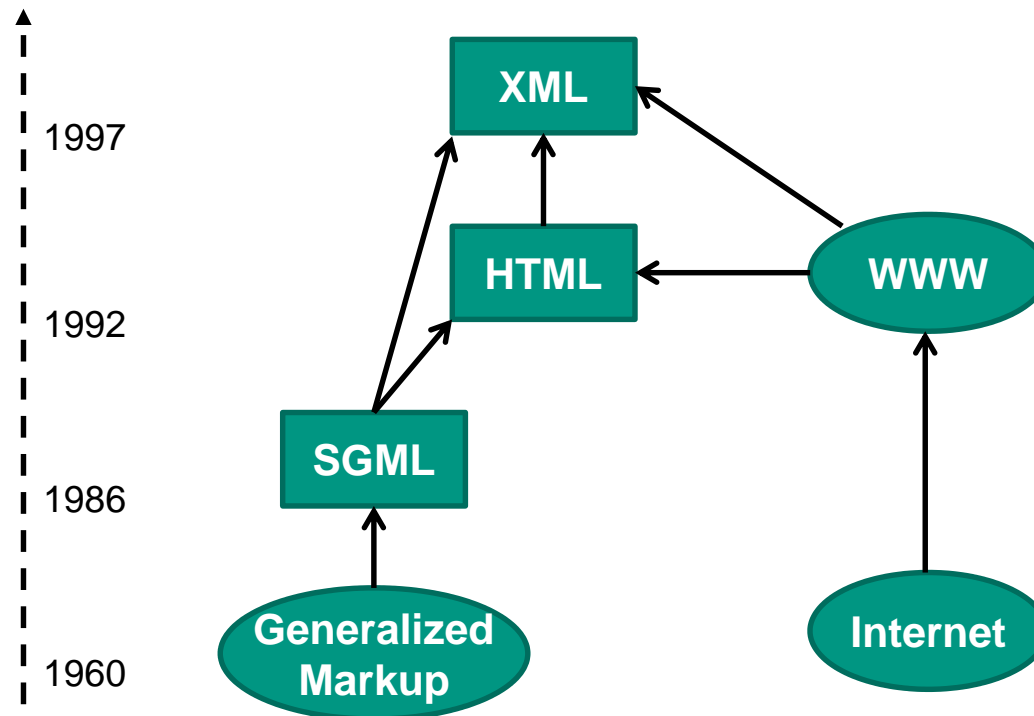
- XML definiert, wie Daten strukturiert in Textdateien gespeichert werden.
- XML-Datenstrukturen sind auch für andere als die ursprüngliche Anwendung „verständlich“.
- Daten können plattformunabhängig zwischen verschiedenen Anwendungen ausgetauscht werden.
- XML ist als Metasprache erweiterbar und ist ebenso wie HTML und XHTML eine Untermenge von SGML.

## ■ Namespaces

- Namensräume, vergleichbar in etwa mit den Packages in Java.
- Werden verwendet damit bestimmte Bezeichner in verschiedenen Zusammenhängen mit unterschiedlicher Bedeutung benutzt werden können.

# Auszeichnungssprachen

- HTML (HyperText Markup Language)
- XML (eXtensible Markup Language)



# XML – eXtensible Markup Language

- Konsequente Trennung zwischen Darstellung und Repräsentation
- Erweiterbar, eigene Tags können definiert werden
- Maschinen les- und prüfbar
- Kein Ersatz für HTML, aber Ergänzung in vielen Bereichen
- Für „normales“ Homepage-Design ist XML relativ uninteressant. HTML und CSS sind dafür längst zu leistungsfähig und angesichts ihrer Verbreitung kaum noch zu verdrängen

- XSL (Extensible Stylesheet Language),  
XSLT (XSL Transformations)
  - Ein XSL-Dokument ist ein XML-Dokument, welches Regeln zur Konvertierung von XML-Dokumenten in andere Formate enthält.
- XML Schema
  - Definiert Art und Struktur von XML-Dokumenten und den enthaltenen Elementen.
  - Wesentlich leistungsfähiger als DTDs und sollen DTDs in Zukunft ersetzen.
- DTD (Document Type Definition)
  - Definition der Dokumentenart und der Bedeutung der Tags bei HTML- und XML-Dokumenten.
  - Auf die DTD wird per DOCTYPE-Anweisung verwiesen.

## ■ DOM (Document Object Model)

- W3C-Standard für Programmierschnittstelle für verschiedene Programmiersprachen zur XML-Verarbeitung.
- DOM konkurriert mit SAX.
- Anders als bei SAX erfolgt bei DOM der XML-Zugriff über einen Objektbaum, was komfortabler ist, aber nicht für große Dokumente geeignet ist.
- Per DOM können XML-Dokumente sowohl gelesen als auch erzeugt werden.

## ■ SAX (Simple API for XML)

- Programmierschnittstelle für verschiedene Programmiersprachen zur XML-Verarbeitung.
- Anders als bei DOM wird bei SAX das XML-Dokument sequentiell durchlaufen, so dass auch sehr große Dokumente bearbeitet werden können.
- Ereignisgesteuert werden bei bestimmten Inhalten vorher (per DocumentHandler-Schnittstelle) registrierte Callbackfunktionen aufgerufen (event-based Parser).
- Anders als bei DOM können mit SAX XML-Dokumente nur gelesen aber nicht erzeugt werden.

# Begriffe und Komponenten zu XML mit Java (1)

- JAXP (Java API for XML Processing)
  - (Auch: Java API for XML Parsing.) Einheitliches API, um unter Java auf XML zuzugreifen. Umfasst DOM, SAX und XSLT und kann verschiedene XML-Parser einbinden.
  - JAXP ist seit J2SE 1.4 in Java enthalten.
- JAXB (Java Architecture for XML Binding)
  - JAXB definiert einen Mechanismus zum Schreiben von Java-Objekten als XML-Dokument (Marshalling) und zum Erzeugen von Java-Objekten aus XML-Dokumenten (Unmarshalling).
- JAXM (Java API for XML Messaging)
  - JAXM definiert einen Mechanismus zum Austausch asynchroner XML-Messages (z.B. für SOAP-Dokumente und ebXML).



- JAX-RPC (Java API for XML-Based Remote Procedure Call)
  - JAX-RPC definiert einen Mechanismus zum Austausch synchroner XML-Messages als Remote Procedure Calls (z.B. für SOAP-RPC), bei denen ähnlich wie bei Funktionsaufrufen auf das Ergebnis gewartet wird.
- JAXR (Java API for XML Registries)
  - JAXR definiert einen Mechanismus zum Veröffentlichen verfügbarer Dienste in einer externen Registry und zur Suche von Diensten in einer solchen Registry (z.B. für UDDI und ebXML Registry).

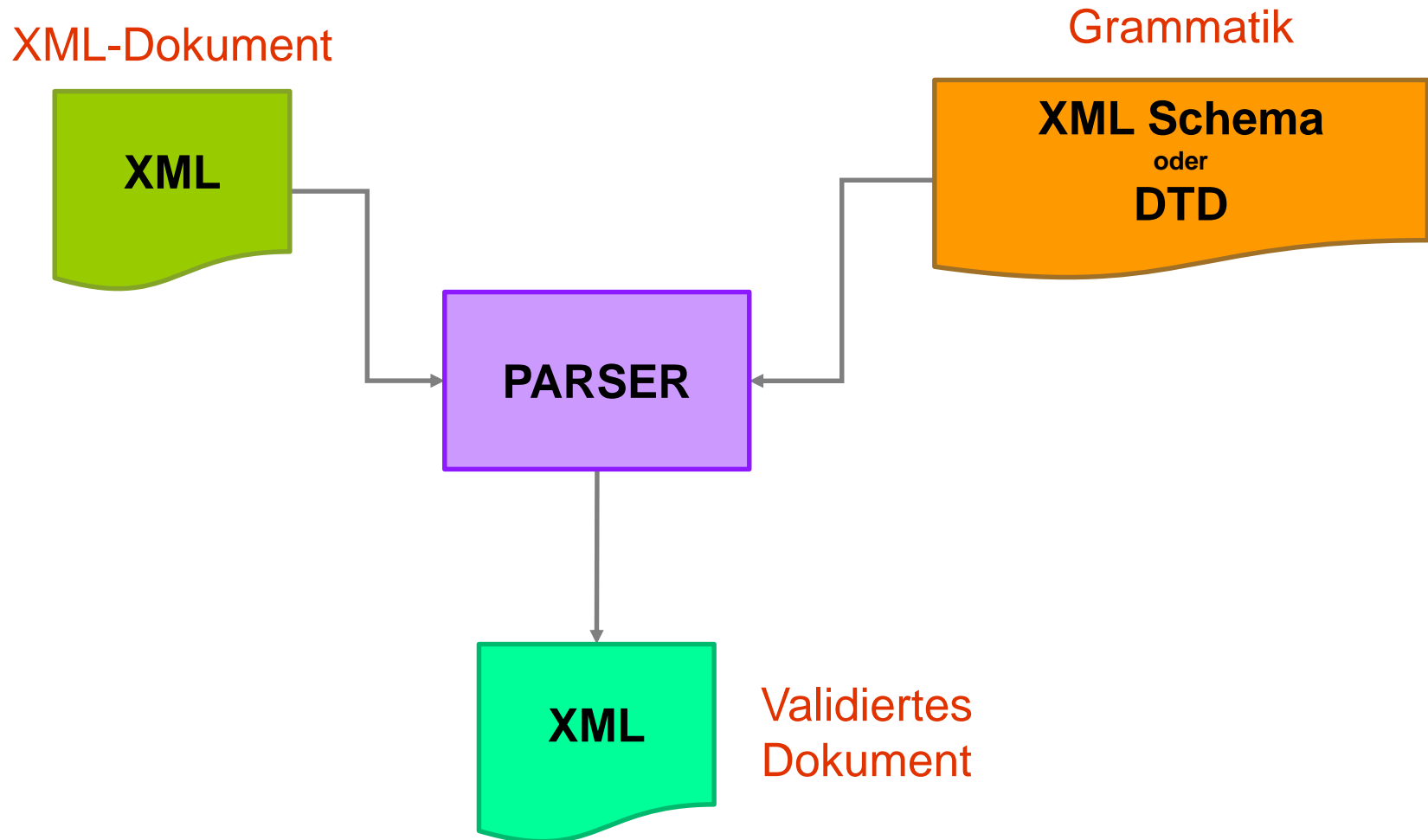
## ■ Xerces, Crimson

- Xerces und Crimson sind XML-Parser, die zum Beispiel über JAXP verwendet werden können.
- Wenn keine besonderen Anforderungen vorliegen, kann einfach das JAXP-API mit dem bereits im JAXP enthaltenen Default-Parser verwendet werden, ohne dass man wissen muss, welcher Parser zu Grunde liegt.

## ■ JDOM (Java DOM), dom4j (DOM for Java)

- JDOM ist eine Java-Bibliothek, die eine an Java angepasste Programmierschnittstelle bietet und einen an Java angepassten Objekt-Tree aus dem XML-Dokument erstellt.
- dom4j ist ebenfalls Open-Source und eine Alternative, die besonders viele Features beinhaltet.

# Gültigkeit von XML-Dokumenten



# Eigenschaften von XML-Dokumenten

- Wohlgeformtheit (wellformed)
  - Dokument syntaktisch korrekt?
    - „Passender“ Aufbau des XML-Dokumentes
    - Korrekte Verschachtelung von Tags
    - Attribute in " "
    - Keine offenen Tags
- Gültigkeit (valid)
  - Wohlgeformtheit und
  - Entspricht einer DTD bzw. einem XML-Schema

# Eigenschaften von XML-Dokumenten

- Der Inhalt eines XML-Dokuments besteht aus strukturierten Elementen, die hierarchisch geschachtelt sind.
- Dazwischen befindet sich der Inhalt, der aus weiteren Elementen (daher hierarchisch) und reinem Text bestehen kann. Die Elemente können Attribute enthalten, die zusätzliche Informationen in einem Element ablegen.

```
<?xml version="1.0" ?>
<party datum="31.12.18">
  <gast name="Albert Angsthase">
    <getraenk>Wein</getraenk>
    <getraenk>Bier</getraenk>
    <zustand ledig="true" nuchtern="false" />
  </gast>
</party>
```



# Die Java-APIs für XML

- DOM (Document Object Model):
  - Liest das gesamte XML-Dokument in eine interne Struktur ein.
  - Standard-DOM unabhängig von der Programmiersprache
- SAX (Simple API for XML Parsing)
  - SAX basiert auf einem Ereignismodell, die XML-Datei wird wie ein Datenstrom gelesen, und für erkannte Elemente wird ein Ereignis ausgelöst. Nachteil: wahlfreier Zugriff auf ein einzelnes Element nicht ohne Zwischenspeicherung möglich ist.
  - zum schnellen Verarbeiten von Daten.
  - SAX ist im Gegensatz zu DOM nicht so speicherhungrig, weil das XML-Dokument nicht vollständig im Speicher abgelegt ist, und daher auch für sehr große Dokumente geeignet.

# Anwendungen von SAX und DOM

- Klassische Anwendungen für SAX sind:
  - Darstellung der XML-Datei in einem Browser
  - Suche nach bestimmten Inhalten
  - Einlesen von XML-Dateien, um eine eigene Datenstruktur aufzubauen
- Für einige Anwendungen ist es erforderlich, die gesamte XML-Struktur im Speicher zu verarbeiten. Für diese Fälle ist eine Struktur, wie sie DOM bietet, notwendig:
  - Sortierung der Struktur oder einer Teilstruktur der XML-Datei
  - Auflösen von Referenzen zwischen einzelnen XML-Elementen
  - Interaktives Arbeiten mit der XML-Datei

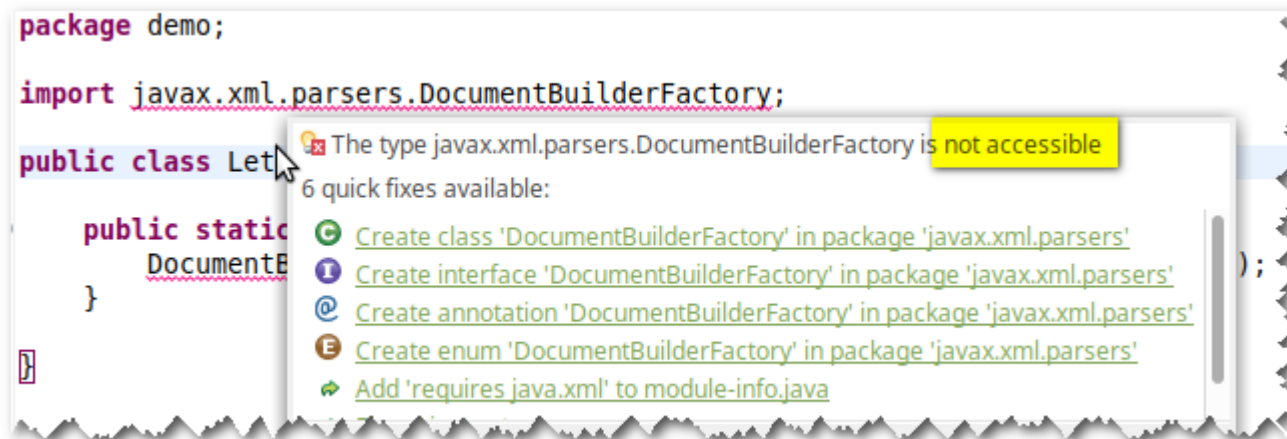
# XML-Dateien mit JAXP verarbeiten

- Die „Java-API for XML-Processing“ (JAXP) ist Bestandteil von Standard-Java (seit Version 1.4).
- JAXP erlaubt XML-formatierte Dateien einzulesen, zu manipulieren und wieder zu schreiben.
- Einfache Speicherung eines Dokuments im Speicher.
- Spezielle Klassen für das Dokument, nämlich Elemente, Attribute und Kommentare.
- JAXP erzeugt eine interne Datenstruktur der XML-Datei.



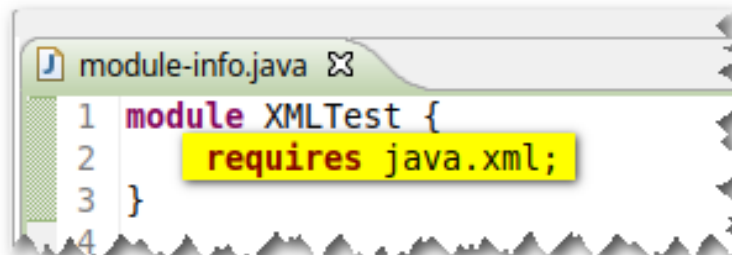
# Verwendung in neuen Java-Versionen (1)

- Klassenbibliothek zur Arbeit mit XML-Dokumenten wird mit dem JDK und JRE bereits mitgeliefert
- Seit Java 9 muss das Programm jedoch **explizit deklarieren**, dass es die XML-Klassen benutzen möchte!



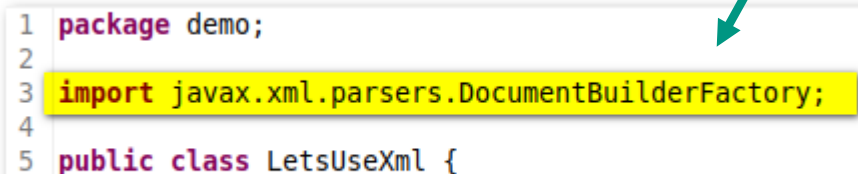
## Verwendung in neuen Java-Versionen (2)

- IDE legt bei Java9+-Projekten normalerweise automatisch eine Datei im Quellcode-Verzeichnis an:  
`module-info.java` (falls nicht: einfach anlegen)
- Dort muss das Modul `java.xml` eingebunden werden:



```
1 module XMLTest {  
2     requires java.xml;  
3 }  
4
```


The screenshot shows a code editor window titled 'module-info.java'. The code defines a module 'XMLTest' which 'requires java.xml;'. The 'requires' line is highlighted in yellow. A green arrow points from this line down to the 'import' line in the code block below.



```
1 package demo;  
2  
3 import javax.xml.parsers.DocumentBuilderFactory;  
4  
5 public class LetsUseXml {
```

The screenshot shows a code editor window with Java code. The 'import javax.xml.parsers.DocumentBuilderFactory;' line is highlighted in yellow. A green arrow points from the 'requires java.xml;' line in the previous code block to this 'import' line.

→ und schon ist der Fehler verschwunden ☺

 **Ab Java 9**

# Das Document-Interface in JAXP

- Dokumente werden über das Interface `org.w3c.dom.Document` definiert.
- Ein Dokument besteht u.a. aus einem `DocumentType`, und einem Wurzelement (mit seinen Unterelementen) und Kommentaren. Methoden von `Document` (Auswahl):

```
// Methode um das Wurzelement zu holen:  
Element getDocumentElement();  
  
// Liefert eine Node-Liste aller enthaltenen Elemente mit  
// dem gegebene Tag-Namen  
NodeList getElementsByTagName(String name);  
  
// Erzeugt ein Element vom angegebenen Typ:  
Element createElement(String tagName);
```

API-Dokumentation zu `Document`:

<http://docs.oracle.com/javase/7/docs/api/org/w3c/dom/Document.html>

# Erzeugen eines Document Objektes (1)

- Ein leeres Dokument im Speicher erstellen :

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;

// ...

DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance();
DocumentBuilder parser = factory.newDocumentBuilder();
Document doc = parser.newDocument();
```



**Kein Exception-  
Handling!**

- Dokument aus einer Datei (mit Pfad `filename`) lesen:

```
DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance();
factory.setNamespaceAware(true);
DocumentBuilder parser = factory.newDocumentBuilder();
Document doc = parser.parse(new File(filename));
```

# Erzeugen eines Document Objektes (2)

## ■ Mit Exception-Handling im Java 1.7-Stil

```
try {  
    DocumentBuilderFactory factory =  
        DocumentBuilderFactory.newInstance();  
    factory.setNamespaceAware(true);  
    DocumentBuilder parser = factory.newDocumentBuilder();  
    String url = "http://example.org/test.xml";  
    Document doc = parser.parse(url);  
    // Ab hier steht der Inhalt als DOM-Baum in "doc" zur Verfügung  
    // ...  
} catch (ParserConfigurationException | SAXException |  
        IOException | DOMException ex) {  
    System.err.println(ex.getMessage());  
}
```

## Erzeugen eines Document Objektes (3)

- Instanzen der Klasse `Document` können nicht per `new` erzeugt werden.
- Es sind immer `DocumentBuilder`, die neue `Document`-Objekte liefern.

Man kann deren Verhalten steuern, z.B. die Validierung auf wohldefinierten XML-Code oder die strenge Beachtung von `NameSpaces` einschalten.

- Methode **parse** von `DocumentBuilder`:

```
Document parse(File f)
```

```
Document parse(InputStream is)
```

```
Document parse(String uri)
```

Parst den Inhalt des angegebenen Files bzw. der angegebenen URI als ein XML-Dokument und liefert ein neues `DOM-Document`-Objekt

# Beispiel: Document aus URL erzeugen

```
public static void main(String[] args) {  
    try {  
        DocumentBuilderFactory factory =  
            DocumentBuilderFactory.newInstance();  
        DocumentBuilder parser = factory.newDocumentBuilder();  
  
        String url = "http://maps.googleapis.com/maps/api/geocode/xml?address=karlsruhe";  
        Document doc = parser.parse(url);  
  
        // Name des Wurzelements ausgeben  
        System.out.println(doc.getDocumentElement().getNodeName());  
    } catch (Exception e) {  
        System.out.println(e.getMessage());  
    }  
}
```

# Das Dokument als XML-Datei ausgeben

- Mit Hilfe der Klassen `javax.xml.transform.Transformer` und `javax.xml.transform.stream.StreamResult` lässt sich der DOM-Baum als XML-Datenstrom (z.B. in eine Datei) ausgeben. Folgende Codezeilen schreiben ein `Document` auf die Konsole:

```
import javax.xml.transform.*; import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;

TransformerFactory tFactory = TransformerFactory.newInstance();
tFactory.setAttribute("indent-number", 4); // Einrückung festlegen

Transformer transformer = tFactory.newTransformer();
transformer.setOutputProperty(OutputKeys.INDENT, "yes"); // schön!

// Das Argument von DOMSource kann ein Document oder
// ein beliebiger Node sein (Unterbaum!)
DOMSource source = new DOMSource(doc);

// StreamResult auf File, einen OutputStream oder Writer
StreamResult result = new StreamResult(System.out);
transformer.transform(source, result);
```



# XML-Dateien einlesen

- Beispiel: Die Datei `party.xml` hat folgenden Inhalt:



```
<?xml version="1.0" ?>
<party datum="31.12.18">
  <gast name="Albert Angsthase">
    <getraenk>Wein</getraenk>
    <getraenk>Bier</getraenk>
    <zustand ledig="true" nuechtern="false" />
  </gast>
  <gast name="Martina Mutig">
    <getraenk>Apfelsaft</getraenk>
    <zustand ledig="true" nuechtern="true" />
  </gast>
  <gast name="Zacharias Zottelig">
  </gast>
</party>
```

Download: <https://www.iai.kit.edu/javav1/data/static/party.xml>

# DOM-Knoten (1)

- Die wichtigsten Knotentypen in DOM sind:
  - Ein **Dokumentknoten** (`org.w3c.dom.Document`) stellt die gesamte Baumstruktur dar;
  - ein **Dokumentfragmentknoten** (`org.w3c.dom.DocumentFragment`) stellt einen Teil der Baumstruktur dar.
  - Ein **Elementknoten** (`org.w3c.dom.Element`) entspricht exakt einem Element in XML
  - Ein **Attributknoten** (`org.w3c.dom.Attr`) entspricht exakt einem Attribut in XML
  - Ein **Textknoten** (`org.w3c.dom.Text`) stellt den textuellen Inhalt eines Elements oder Attributs (dessen Value) dar
  - Ein **Kommentarknoten** (`org.w3c.dom.Comment`) repräsentiert den Inhalt eines XML-Kommentars, d.h. den Inhalt zwischen '`<!--`' und '`-->`'.

## DOM-Knoten (2)

- Das Interface `org.w3c.dom.Node` bildet das Superinterface all dieser Knoten.
  - `short getNodeType()` liefert den Typ des Knotens entsprechend vordefinierten Werten:

**ATTRIBUTE\_NODE**, **CDATA\_SECTION\_NODE**, **COMMENT\_NODE**,  
**DOCUMENT\_FRAGMENT\_NODE**, **DOCUMENT\_NODE**, **DOCUMENT\_TYPE\_NODE**,  
**ELEMENT\_NODE**, **ENTITY\_NODE**, **ENTITY\_REFERENCE\_NODE**,  
**NOTATION\_NODE**, **PROCESSING\_INSTRUCTION\_NODE**, **TEXT\_NODE**

## DOM-Knoten (3)

- Das Interface `org.w3c.dom.Node` definiert u.a. folgende weiteren Methoden:

- `NamedNodeMap` **`getAttributes()`**
- `NodeList` **`getChildNodes()`**
- `Node` **`getFirstChild()`**, **`getLastChild()`**
- `Node` **`getNextSibling()`**, **`getPreviousSibling()`**
- `String` **`getNodeName()`**
- `String` **`getNodeValue()`**
- `short` **`getNodeType()`**
- `String` **`getTextContent()`**
- `Node` **`appendChild(Node newChild)`**
- `Node` **`removeChild(Node oldChild)`**
- `void` **`setTextContent(String textContent)`**
- `void` **`setNodeValue(String nodeValue)`**
- `Node` **`getParentNode()`**

## DOM-Knoten (4)

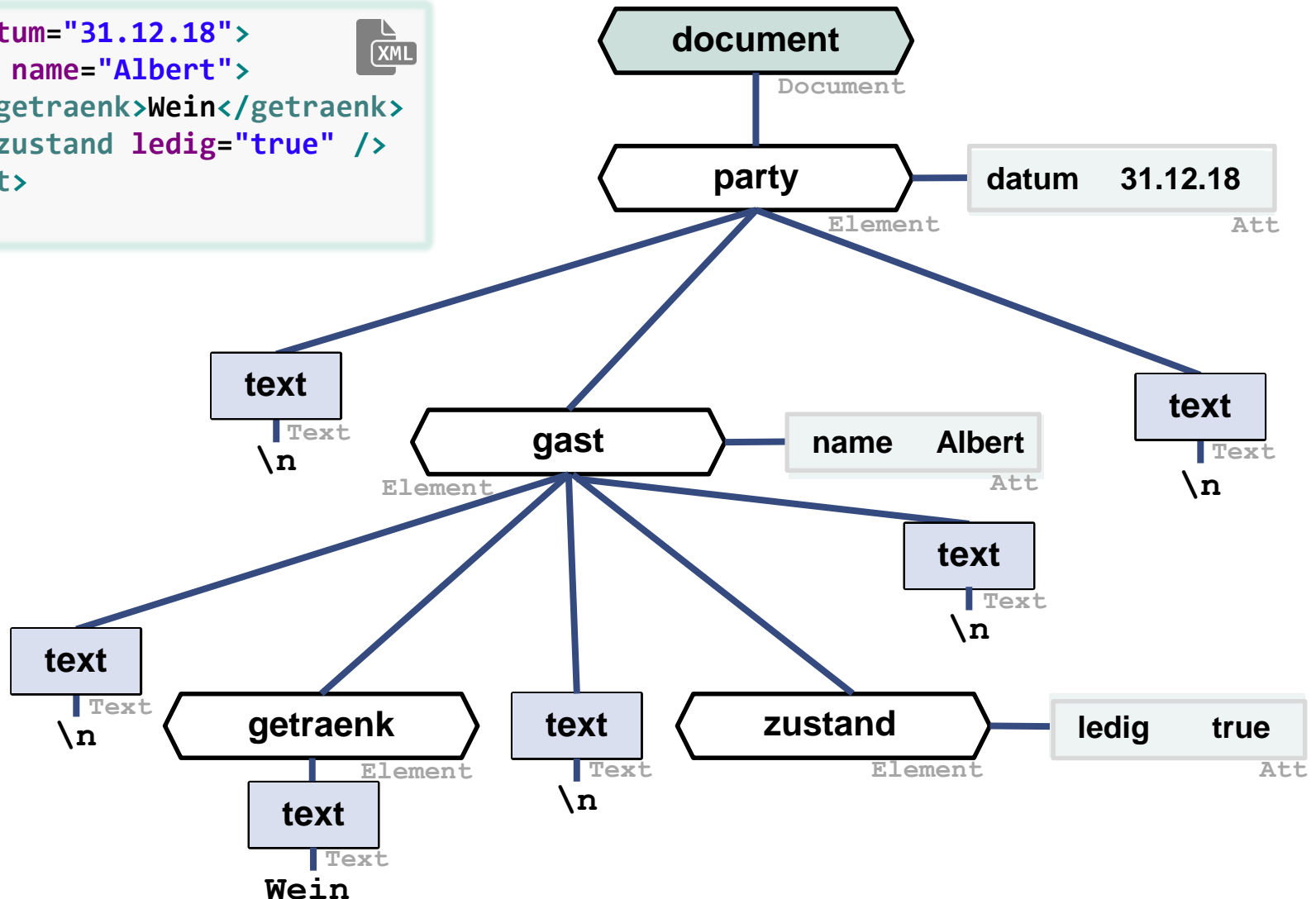
- Das Interface `org.w3c.dom.NodeList` bietet Zugriff auf die Elemente einer Menge von Knoten (z.B. die Kinder eines Knotens):
  - `int getLength()` liefert die Anzahl der Knoten
  - `Node item(int n)` liefert den n-ten Knoten dieser Liste
  
- Das Interface `org.w3c.dom.NamedNodeMap` bietet Zugriff auf die Elemente, die über einen Namen angesprochen werden können, z.B. Attribute
  - `Node getNamedItem(String name)` liefert den Knoten mit dem Namen `name`
  - `int getLength()` liefert die Anzahl der Knoten
  - `Node item(int n)` liefert den n-ten Knoten dieser Liste

# nodeName, nodeValue und Attributes für vers. Knoten

Interface	nodeName	nodeValue	Attributes
<b>Attr</b>	Name des Attributs	Wert des Attributs	null
<b>CDATASection</b>	"#cdata-section"	Inhalt der CDATA-Sektion	null
<b>Comment</b>	"#comment"	Inhalt des Kommentars	null
<b>Document</b>	"#document"	null	null
<b>DocumentFragment</b>	"#document-fragment"	null	null
<b>DocumentType</b>	Name des „document type“	null	null
<b>Element</b>	Name des Tags	null	NamedNodeMap
<b>Entity</b>	Name des Entity	null	null
<b>EntityReference</b>	Name des referenzierten Entity	null	null
<b>Notation</b>	notation name	null	null
<b>ProcessingInstruction</b>	Target	Gesamter Inhalt ohne das Target	null
<b>Text</b>	"#text"	Inhalt des Text-Knotens	null

# Beispiel-DOM-Baum (vereinfacht)

```
<party datum="31.12.18">
  <gast name="Albert">
    <getraenk>Wein</getraenk>
    <zustand ledig="true" />
  </gast>
</party>
```



# Informationen im DOM-Baum ermitteln (1)

```
// Erzeugen eines Dokuments anhand der Datei party.xml
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder parser = factory.newDocumentBuilder();
Document doc = parser.parse("https://www.iai.kit.edu/javav1/data/static/party.xml");

// Lesen des Wurzelements des DOM-Dokuments doc
org.w3c.dom.Element party = doc.getDocumentElement();

// Den ersten Gast auf der Party holen:
org.w3c.dom.Node albert = party.getElementsByTagName("gast").item(0);

// Wenn wir eine Gästeliste der Party haben wollen:
// getElementsByTagName nur für Knoten vom Typ Element
NodeList gaeste = party.getElementsByTagName("gast");
for (int i = 0; i < gaeste.getLength(); i++) {
    System.out.println( gaeste.item(i).getAttributes()
                        .getNamedItem("name").getNodeValue());
}
```



## Informationen im DOM-Baum ermitteln (2)

```
// Wenn wir wissen wollen, was Albert trinkt:  
Node getraenk = null;  
for (int i = 0; i < albert.getChildNodes().getLength(); i++) {  
    if (albert.getChildNodes().item(i).getNodeName().equals("getraenk")) {  
        getraenk = albert.getChildNodes().item(i);  
        System.out.println(getraenk.getTextContent());  
    }  
}
```



## Informationen im DOM-Baum ermitteln (3)

- Da die Methode `item(n)` lediglich `Node`-Objekte liefert, kann man deren Typ Abfragen und sie in `Element`-Objekte casten:

```
// Den ersten Gast auf der Party holen:  
Node albert = party.getElementsByTagName("gast").item(0);  
  
Element albertAsElement = null;  
if (albert.getNodeType() == Node.ELEMENT_NODE) {  
    albertAsElement = (Element) albert;  
}
```

- oder direkt casten ...

```
// Den ersten Gast auf der Party holen:  
Element albertAsElement = (Element)  
    party.getElementsByTagName("gast").item(0);
```

... da `getElementsByTagName` nur Elementknoten liefert

## Informationen im DOM-Baum ermitteln (4)

- Da es keine Methode gibt, um ein bestimmtes Element-Kind eines `Node` zu liefern, kann man sich eine eigene Methode schreiben:

```
// Liefert das erste Kind-Element von n mit Namen name
public static Element getNamedChildElement(Node n, String name) {
    return getNamedChildElement(n, name, 0);
}

// Liefert das n-te Kind-Element von n mit Namen name, Zählung beginnt bei 0
public static Element getNamedChildElement(Node n, String name, int count) {
    for (int i = 0; i < n.getChildNodes().getLength(); i++) {
        Node ithChild = n.getChildNodes().item(i);
        if (ithChild.getNodeType() == Node.ELEMENT_NODE &&
            ithChild.getNodeName().equals(name)) {
            if (count == 0) {
                return (Element) ithChild;
            }
            count--;
        }
    }
    return null;
}
```

# Neue Elemente einfügen und ändern

- Ein weiteres Getränk zu Gast Albert hinzufügen:

```
Element party = doc.getDocumentElement();
Node albert = party.getElementsByTagName("gast").item(0);
Node water = doc.createElement("getraenk");
water.settextContent("Wasser");
albert.appendChild(water);
```

- Albert will in Zukunft keinen Wein mehr trinken:

```
Node wine = null;
for (int i = 0; i < albert.getChildNodes().getLength(); i++) {
    Node ithChild = albert.getChildNodes().item(i);
    if (ithChild.getNodeName().equals("getraenk")) {
        if (ithChild.getTextContent().equals("Wein")) {
            wine = ithChild;
        }
    }
}
if (wine != null){
    albert.removeChild(wine);
}
```

# Attributinhalte lesen und ändern

## ■ Lesen des Name-Attributs von Albert:

```
Element party = doc.getDocumentElement();
Node albert = party.getElementsByTagName("gast").item(0);
System.out.println(albert.getAttribute("name").getNodeValue());
```

## ■ Martina schnappt sich Albert, wenn er noch ledig ist:

```
Node state = null;
for (int i = 0; i < albert.getChildNodes().getLength(); i++) {
    if (albert.getChildNodes().item(i).getNodeName().equals("zustand")) {
        state = albert.getChildNodes().item(i);
    }
}
if (state != null) {
    // Attr. lesen
    String val = state.getAttribute("ledig").getNodeValue();
    if ( "true".equals(val) ) {
        // Attr. setzen: entsprechend bei Martina!
        state.getAttribute("ledig").setNodeValue("false");
    }
}
```

# DOM-Baum erzeugen

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder parser = factory.newDocumentBuilder();
Document doc = parser.newDocument();
Element fib = doc.createElement("Fibonacci_Numbers");
doc.appendChild(fib);
int low = 1;
int high = 1;

// Erzeugen des DOM-Baumes
for (int i = 1; i <= 10; i++) {
    Element fibonacci = doc.createElement("fibonacci");
    fibonacci.setAttribute("index", String.valueOf(i));
    fibonacci.setTextContent(String.valueOf(low));
    fib.appendChild(fibonacci);
    int temp = high;
    high += low;
    low = temp;
}
```

# Erzeugter DOM-Baum (in XML)

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Fibonacci_Numbers>
  <fibonacci index="1">1</fibonacci>
  <fibonacci index="2">1</fibonacci>
  <fibonacci index="3">2</fibonacci>
  <fibonacci index="4">3</fibonacci>
  <fibonacci index="5">5</fibonacci>
  <fibonacci index="6">8</fibonacci>
  <fibonacci index="7">13</fibonacci>
  <fibonacci index="8">21</fibonacci>
  <fibonacci index="9">34</fibonacci>
  <fibonacci index="10">55</fibonacci>
</Fibonacci_Numbers>
```

