Netzprogrammierung
XML Dokumente und ihre Verarbeitung

Prof. Dr.-Ing. Robert Tolksdorf
Freie Universität Berlin
Institut für Informatik
Netzbasierte Informationssysteme
mailto: tolk@inf.fu-berlin.de
http://www.robert-tolksdorf.de

Überblick

1. XML Dokumententypen
2. XML Parser DOM und SAX

Standardisierung

- prinzipieller Aufbau von Dokumenten: Welche Elemente/Attribute dürfen wo verwendet werden?
- Datentypen der Inhalte: Welche Inhalte sind erlaubt?

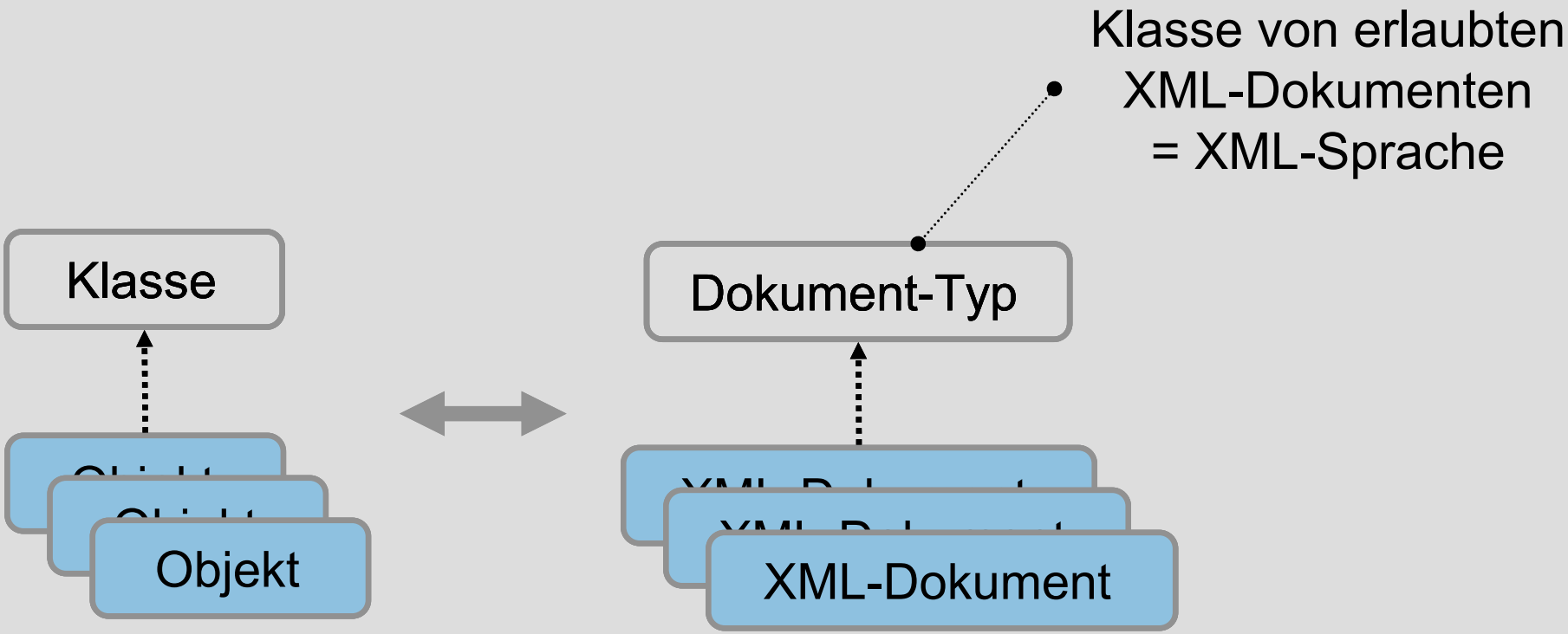
```
<Book>
   <Title> PCDATA </Title>
   <Author> PCDATA </Author>
   <Date> PCDATA </Date>
   <ISBN> PCDATA </ISBN>
   <Publisher> PCDATA </Publisher>
</Book>
```

- konkrete Inhalte werden *nicht* beschrieben

⇨ Klasse von erlaubten XML-Dokumenten

- auch Dokument-Typ, XML-Sprache oder Anwendung von XML genannt

Dokument-Typ



- Dokument-Typ kann mit einer DTD oder einem XML-Schema definiert werden.

Document Type Definitions (DTDs)

Wie könnte eine DTD hierfür aussehen?


```
<BookStore>
  <Book>
      <Title>My Life and Times</Title>
      <Author>Paul McCartney</Author>
      <Date>July, 1998</Date>
      <ISBN>94303-12021-43892</ISBN>
      <Publisher>McMillin Publishing</Publisher>
  </Book>
</BookStore>
```

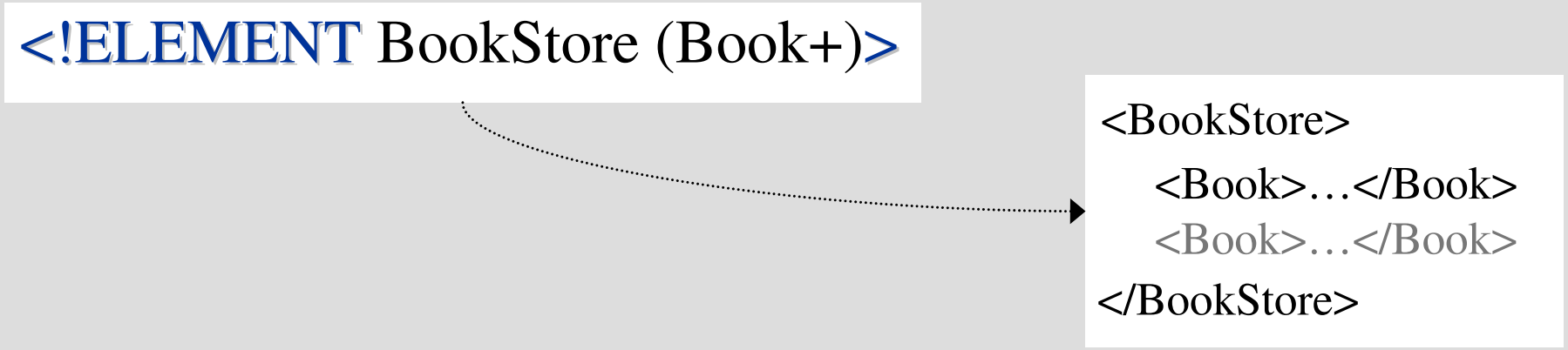
- BookStore soll mindestens ein Buch enthalten.
- ISBN optional
- alle anderen Kind-Elemente obligatorisch

Die DTD für das Beispiel-Dokument


```
<!ELEMENT BookStore (Book+)>
<!ELEMENT Book (Title, Author, Date, ISBN?, Publisher)>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Author (#PCDATA)>
<!ELEMENT Date (#PCDATA)>
<!ELEMENT ISBN (#PCDATA)>
<!ELEMENT Publisher (#PCDATA)>
```

ähnelt einer regulären Grammatik

Deklaration von BookStore



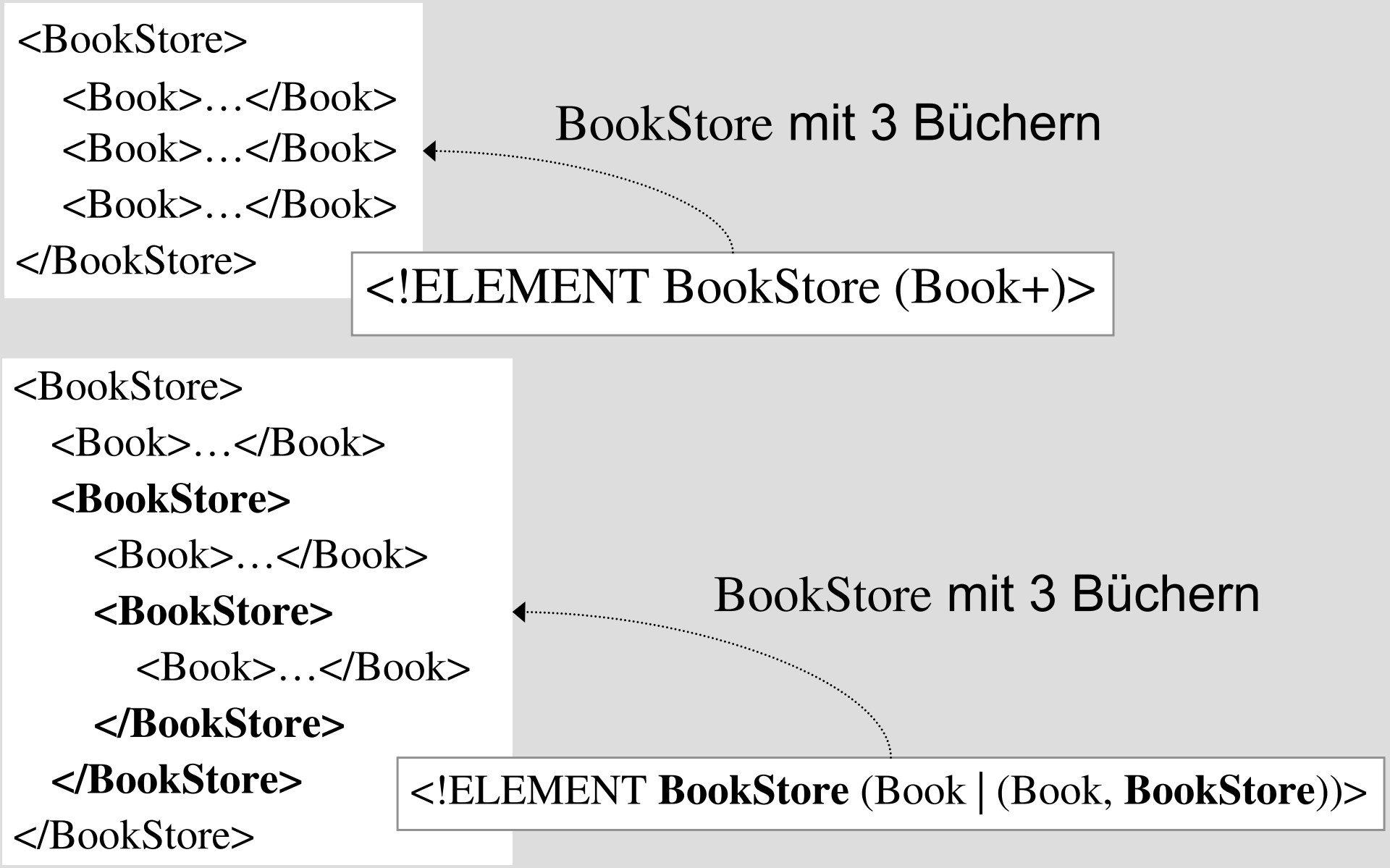
- BookStore hat mindestens ein Kind-Element Book.
- + bezeichnet n Wiederholung des vorstehenden Elementes mit n ≥ 1.
- * bezeichnet n Wiederholung mit n ≥ 0.
- Außer Book darf BookStore keine anderen Kind-Elemente haben.
- Element-Deklaration genannt

Rekursive Deklarationen


```
<!ELEMENT BookStore (Book | (Book, BookStore))>
```

- Bookstore besteht aus genau einer der folg. Alternativen:
  - genau ein Kind-Element Book
  - zwei Kind-Elemente: Book und BookStore
- | bezeichnet **Auswahl**: genau eine der beiden Alternativen
- , bezeichnet **Sequenz** von Elementen.
- Beachte: Rekursive Deklaration *nicht* äquivalent zur vorherigen, iterativen Definition!

Rekursive vs. iterative Deklaration



Deklaration von Book


```
<!ELEMENT Book (Title, Author, Date, ISBN?, Publisher)>
```

- Title, Author, Date, ISBN und Publisher (in dieser Reihenfolge) Kind-Elemente von Book.
- außer diesen keinen anderen Kind-Elemente

```
<Book>
    <Title>…</Title>
    <Author>…</Author>
    <Date>…</Date>
    <ISBN>…</ISBN>
    <Publisher>…</Publisher>
</Book>
```

- ? bedeutet, dass Element optional ist.

Deklaration von Title etc.


```
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Author (#PCDATA)>
<!ELEMENT Date (#PCDATA)>
<!ELEMENT ISBN (#PCDATA)>
<!ELEMENT Publisher (#PCDATA)>
```

```
<Title>My Life and Times</Title>
<Author>Paul McCartney</Author>
<Date>July, 1998</Date>
<ISBN>94303-12021-43892</ISBN>
<Publisher>McMillin Publishing</Publisher>
```

- **#PCDATA**: unstrukturierter Inhalt ohne reservierte Symbole < und &.

Datentypen für Element-Inhalte

nur drei verschiedene Datentypen:

1. **#PCDATA**: unstrukturierter Inhalt ohne reservierte Symbole < und &.
2. **EMPTY**: leerer Inhalt, Element kann aber Attribute haben

   <!ELEMENT br EMPTY> →
3. **ANY**: beliebiger Inhalt (strukturiert, unstrukturiert, gemischt oder leer)

   <!ELEMENT title ANY>

- Beachte: Datentypen wie INTEGER oder FLOAT stehen *nicht* zur Verfügung.

Verschachtelungen

- beliebige Verschachtelung von Sequenz, Auswahl |, ?, *, + und Rekursion erlaubt
- Beispiel:

```
<!ELEMENT Chap (Title, (Text | Chap)+)>
<!ELEMENT Text ANY>
<!ELEMENT Title (#PCDATA)>
```

```
<Chap>
  <Title>Kap1</Title>
  <Text>…</Text>
  <Chap>
    <Title>Kap1.1</Title>
    <Text>…</Text>
  </Chap>
  <Text>…</Text>
  <Chap>
    <Title>Kap1.2</Title>
    <Text>…</Text>
  </Chap>
</Chap>
```

Deklaration von Attributen

```
<!ATTLIST BookStore
                    version CDATA #IMPLIED>
```

```
<BookStore version="1.0">
   …
</BookStore>
```

- Element BookStore hat Attribut version.
- Außer version hat BookStore keine weiteren Attribute.
- Attribut-Deklaration genannt
- CDATA: Attribut-Wert ist String ohne <, &, ' und "
- Beachte: nicht verwechseln mit <![CDATA[…]]>
- daher Entity References für <, &, ' und " verwenden!

Deklaration von Attributen

```
<!ATTLIST BookStore
                    version CDATA #IMPLIED "1.0">
```

- #IMPLIED: Attribut optional
- "1.0": Standard-Wert des Attributes

⇨ wenn Attribut nicht vorhanden, fügt XML-Parser Attribut mit Standard-Wert hinzu

statt #IMPLIED auch möglich:

- #REQUIRED: Attribut obligatorisch
- #FIXED: Attribut hat immer den gleichen Wert.

Aufzählungstypen

```
<!ATTLIST Author
                    gender (male | female) "female">
```

- hier statt CDATA **Aufzählungstyp**:
- Attribut gender hat entweder den Wert male oder female (Aufzählungstyp).
- "female" ist Standard-Wert von gender.

Datentypen für Attribute

Zusätzlich zu CDATA (Strings) und Aufzählungstypen:

- NMTOKEN: String, der Namenskonventionen von XML entspricht
- ID: eindeutiger Bezeichner, der Namenskonventionen von XML entspricht
- IDREF: Referenz auf einen eindeutigen Bezeichner

ID/IDREF

```
<!ATTLIST Author
                key ID #IMPLIED
                keyref IDREF #IMPLIED>
```

- Wert des Attributes key muss *eindeutig* sein:

  Zwei Attribute vom Typ ID dürfen niemals gleichen Wert haben.
- Wert des Attributes keyref muss *gültige Referenz* sein:

  Wert von keyref muss als Wert eines Attributes vom Typ ID erscheinen.

Beispiel

```
<BookStore>
  <Book>
      <Title>Text</Title>
      <Author key="k1">Text</Author>
      <Date>Text</Date>
      <Publisher>Text</Publisher>
  </Book>
  <Book>
      <Title>Text</Title>
      <Author keyref="k1"/>
      <Date>Text</Date>
      <Publisher>Text</Publisher>
  </Book>
</BookStore>
```

Wert k1 muss eindeutig sein: kein anderes Attribut vom Typ ID darf diesen Wert haben.

Referenz k1 muss existieren: ein Attribut vom Typ ID muss den Wert k1 haben.

Festlegung des Dokument-Typs

- Prozessorinstruktion direkt nach der XML-Deklaration:

  <!DOCTYPE *Wurzel-Element* SYSTEM "*DTD*">

- legt Wurzel-Element und Dokument-Typ fest

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE BookStore SYSTEM "Bookstore.dtd">
<BookStore>
    …
</BookStore>
```

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE Book SYSTEM "Bookstore.dtd">
<Book>
    …
</Book>
```

Wohlgeformheit vs. Zulässigkeit

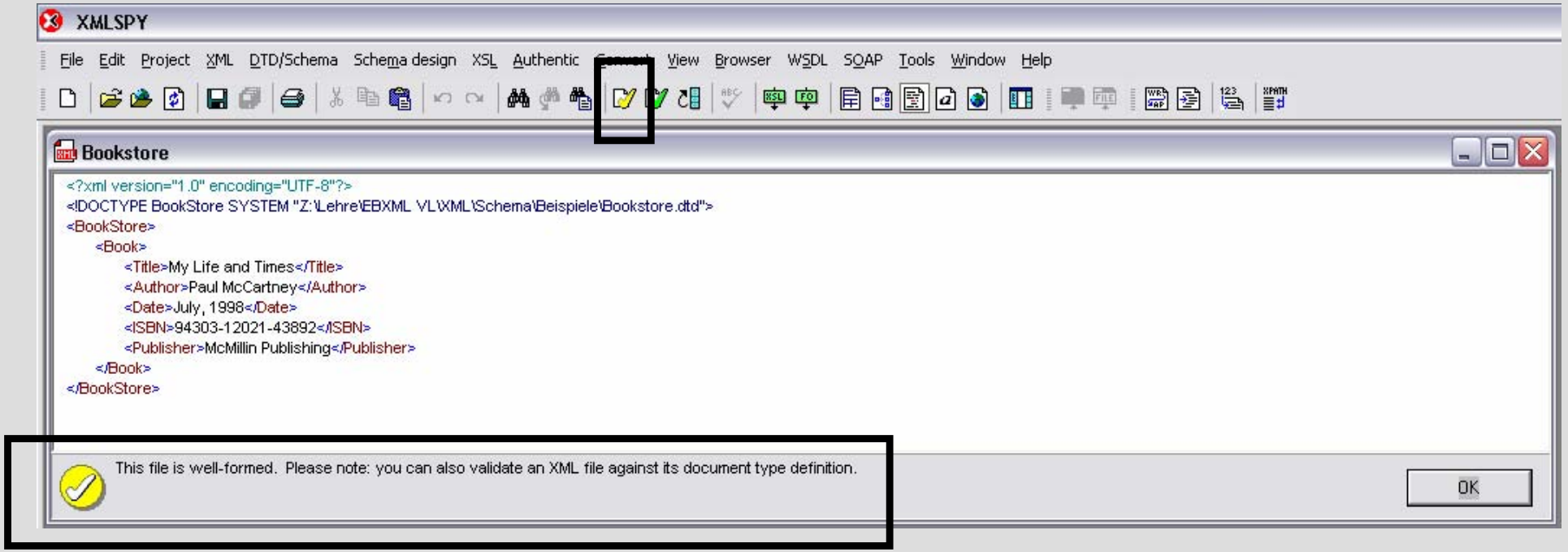
wohlgeformt (well formed)

- XML-Dokument entspricht syntaktischen Regeln von XML

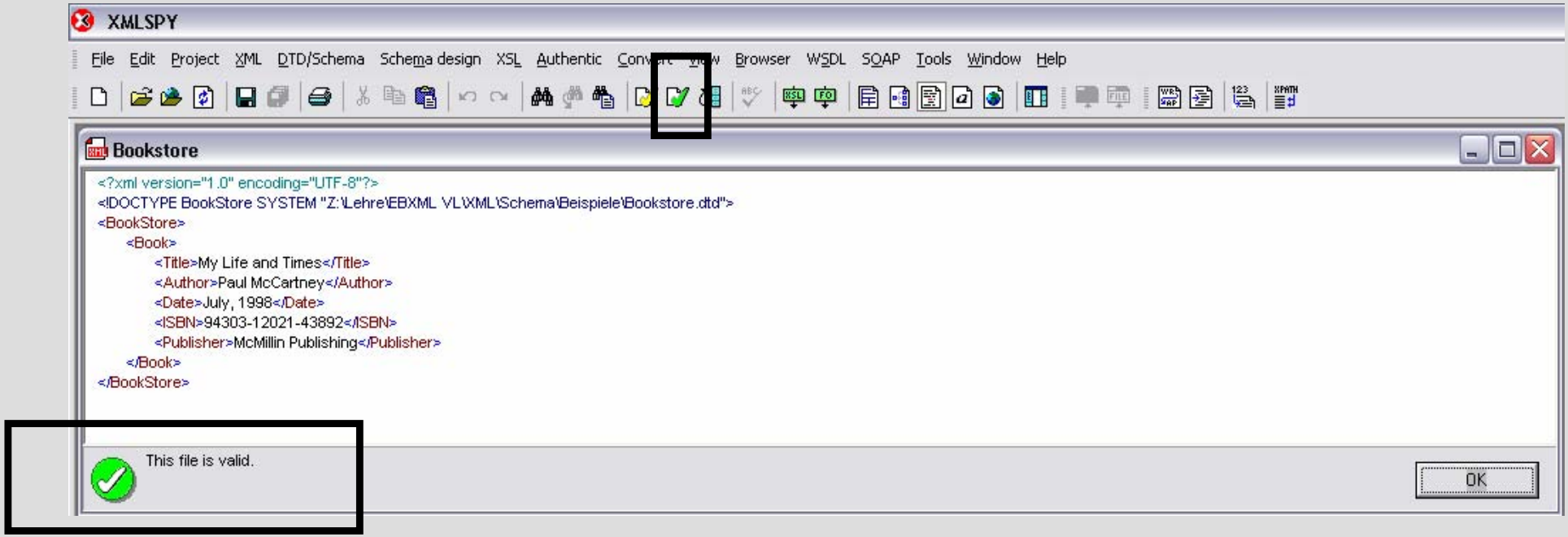
zulässig (valid) bzgl. einer DTD

1. Wurzel-Element des XML-Dokumentes ist in der DTD deklariert und
2. Wurzel-Element hat genau die Struktur, wie sie in der DTD festgelegt ist.

Prüfung der Wohlgeformtheit



Prüfung der Zulässigkeit



Nachteile von DTDs

- keine XML-Syntax, eigener Parser nötig
- nur sehr wenige Datentypen, insbesondere für Element-Inhalte
- keine eigenen Datentypen definierbar
- keine Namensräume:

  DTDs können nur dann kombiniert werden, wenn es *keine* Namenskonflikte gibt!
- keine Vererbungshierarchien, nicht objekt-orientiert

Und noch ein Nachteil

- Sequenzen einfach zu definieren:

  <!ELEMENT Book (Title, Author)>

⇨ starre Struktur in XML-Dokumenten

- soll Reihenfolge der Kind-Elmente egal sein, müssen alle Permutationen explizit aufgezählt werden:

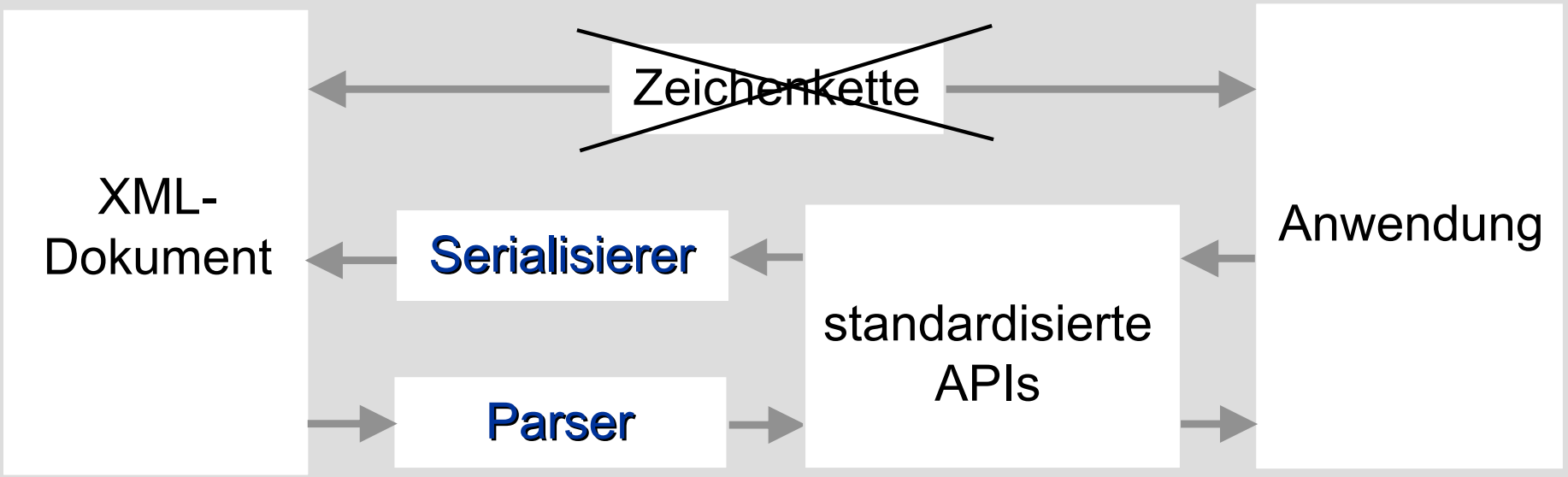
  <!ELEMENT Book ((Title, Length) | (Length, Title))>

- nicht praktikabel: bei *n* Kind-Elementen *n*! Permutationen

- *XML Schema stellt Mechanismus zur typisierten Beschreibung von Dokumentenschemata bereit*
- *Vorlesung Schild: XML-Technologien*

XML-Parser

Grundlegende Architektur



Parser

- analysiert XML-Dokument und erstellt Parse-Baum mit Tags, Text-Inhalten und Attribut-Wert-Paaren als Knoten

Serialisierer

- generiert aus Datenstruktur XML-Dokument

Kategorien von Parser

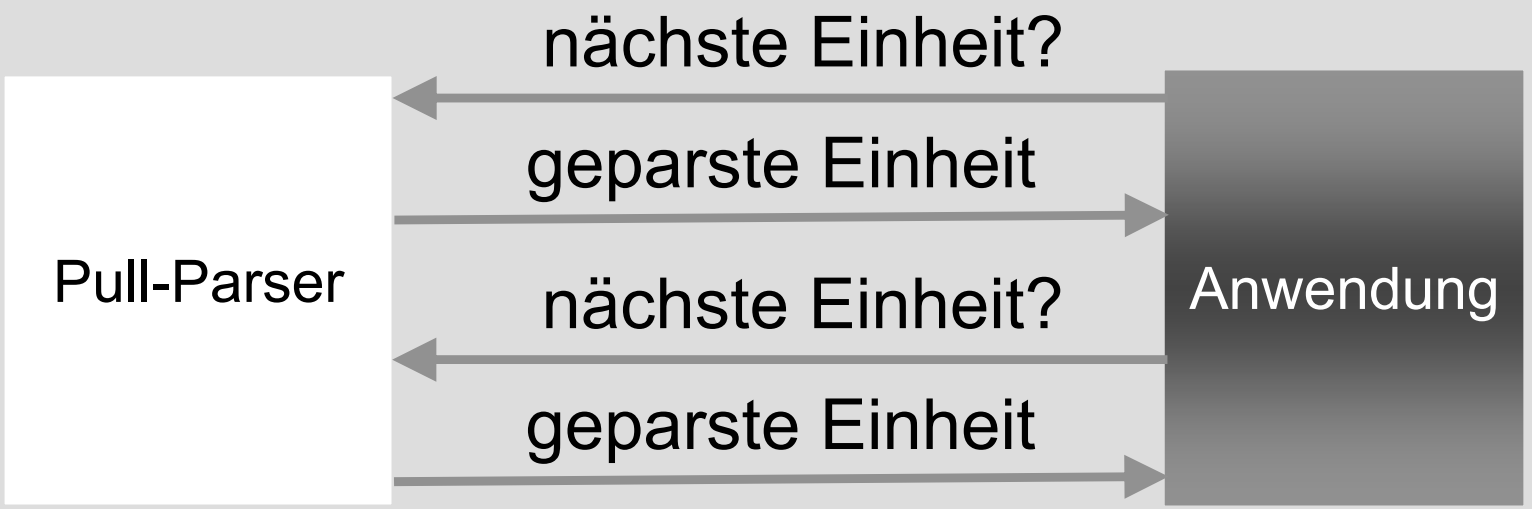
Pull- vs. Push-Parser

- Wer hat Kontrolle über das Parsen: die Anwendung oder der Parser?

Einschritt- vs. Mehrschritt-Parser (*one step vs. multi step*)

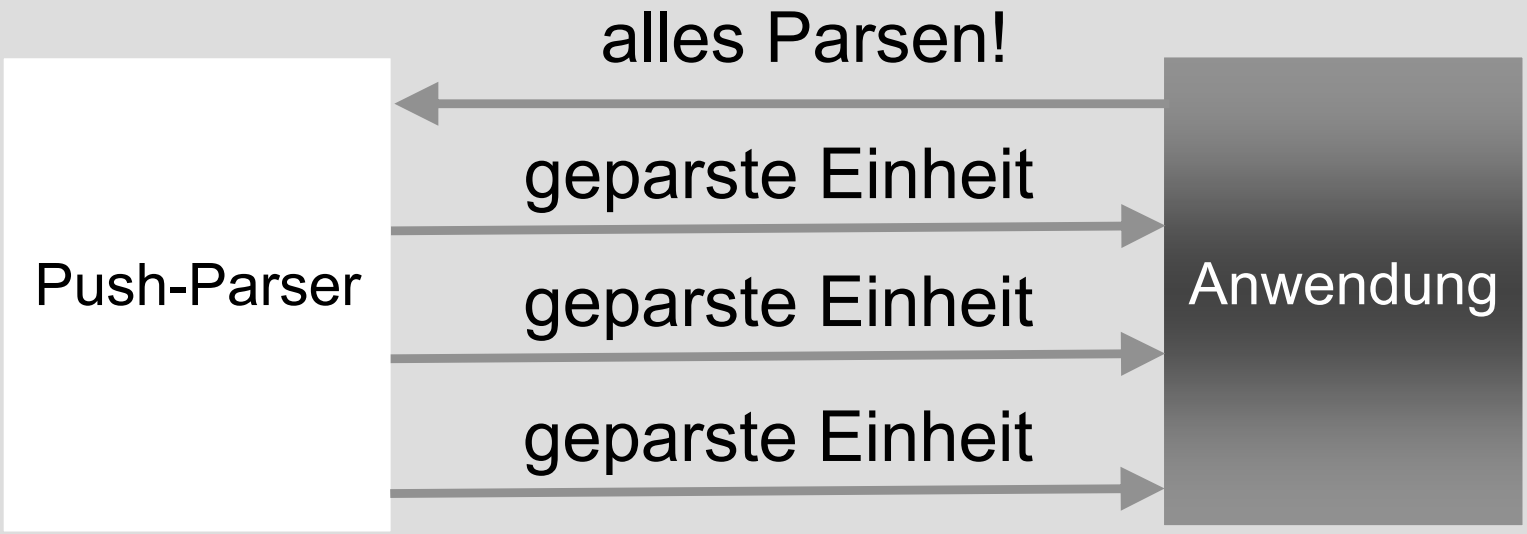
- Wird das XML-Dokument in einem Schritt vollständig geparst oder Schritt für Schritt?
- Beachte: Kategorien unabhängig voneinander, können kombiniert werden

Pull-Parser



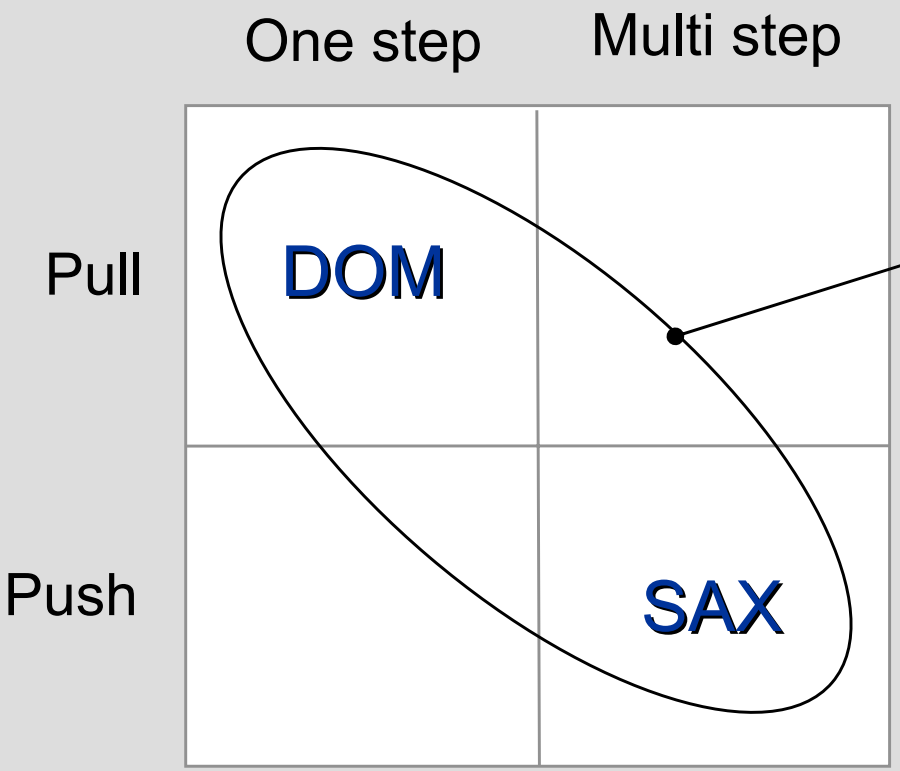
- Anwendung hat Kontrolle über das Parsen.
- Analyse der nächsten syntaktischen Einheit muss aktiv angefordert werden.
- Beachte: „Pull” bezieht sich auf die Perspektive der Anwendung.

Push-Parser



- Parser hat Kontrolle über das Parsen.
- Sobald der Parser eine syntaktische Einheit analysiert hat, übergibt er die entsprechende Analyse.
- Beachte: „Push” bezieht sich wiederum auf die Perspektive der Anwendung.

XML-Parser



- JAXP: Java API for XML Processing
- JAXP 1.3 in J2SE 5.0 enthalten

- DOM: Document Object Model
- SAX: Simple API for XML

SAX-Parser

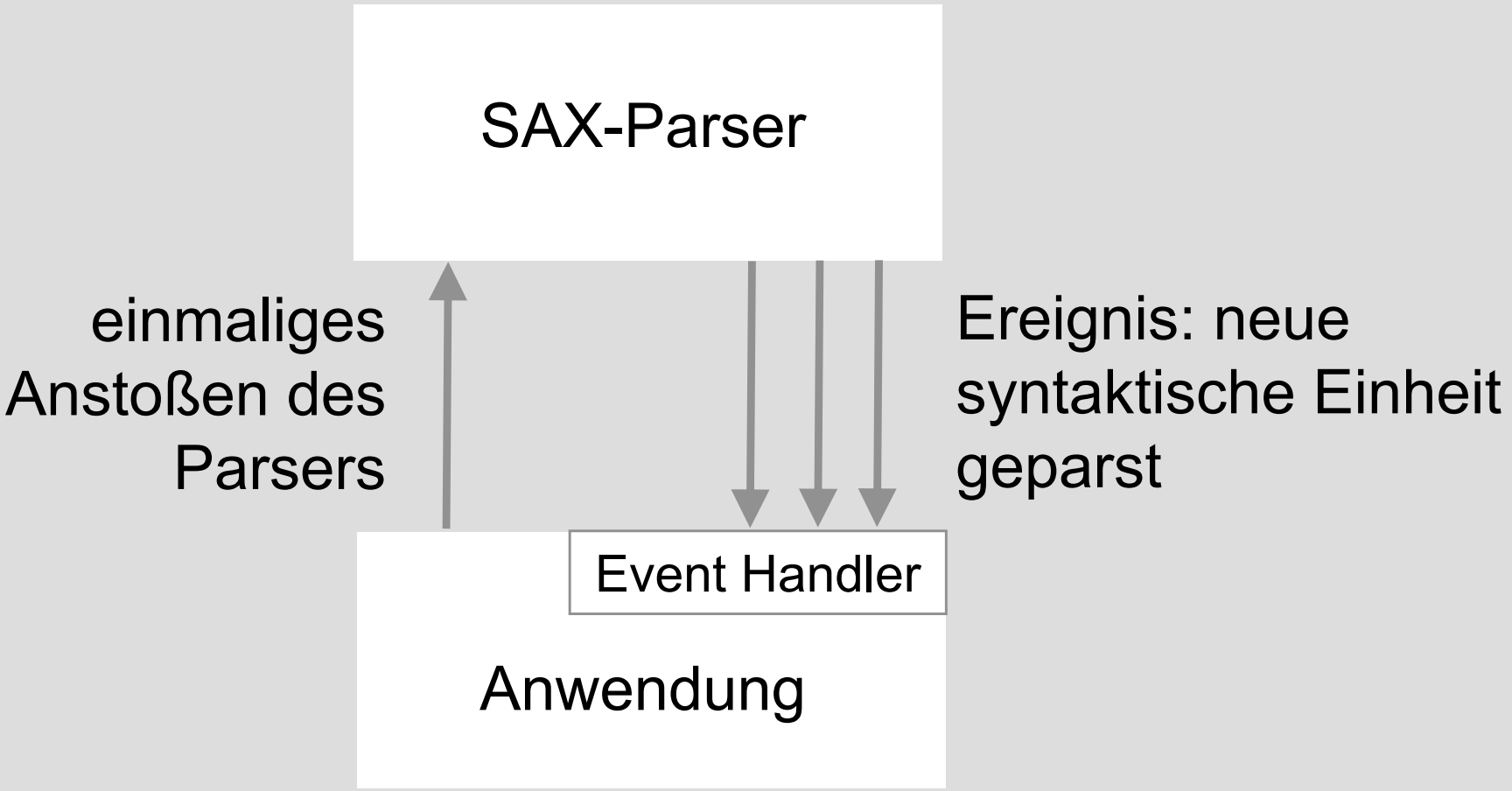
SAX: Simple API for XML

- Mehrschritt-Push-Parser für XML
- standardisiertes API
- ursprünglich nur Java-API
- inzwischen werden aber auch viele andere Sprachen unterstützt: C, C++, VB, Pascal, Perl
- kein W3C-Standard, sondern *de facto* Standard

➔ http://www.saxproject.org/

- auch in MSXML integriert

Ereignisbasiertes Parsen



Beispiel


```
<priceList>
  <coffee>
      <name>
            Mocha Java
      </name>
      <price>
            11.95
      </price>
  </coffee>
</priceList>
```

Parser ruft startElement(…,priceList,…) auf.
Parser ruft startElement(…,coffee,…) auf.
Parser ruft startElement(…,name,…) auf.
Parser ruft characters("Mocha Java",…) auf.
Parser ruft endElement(…,name,..) auf.
Parser ruft startElement(…,price,…) auf.
Parser ruft characters("11.95",…) auf.
Parser ruft endElement(…,price,…) auf.
Parser ruft endElement(…,coffee,…) auf.
Parser ruft endElement(…,priceList,…) auf.

- Ereignisfluss: Sobald Einheit geparst wurde, wird Anwendung benachrichtigt.
- Beachte: Es wird *kein* Parse-Baum aufgebaut!

Callback-Methoden

- Methoden des Event-Handlers (also der Anwendung), die vom Parser aufgerufen werden
- für jede syntaktische Einheit eigene Callback-Methode, u.a.:
  - startDocument und endDocument
  - startElement und endElement
  - characters
  - processingInstruction

DefaultHandler

- Standard-Implementierung der Callback-Methoden: tun jeweils nichts!
- können aber überschrieben werden

startElement

```
public void startElement(java.lang.String uri,
                         java.lang.String localName,
                         java.lang.String qName,
                         Attributes attributes)
        throws SAXException
```

- uri: Namensraum-Bezeichner oder leerer String
- localName: lokaler Name ohne Präfix oder leerer String
- qName: Name mit Präfix oder leerer String
- attributes: zu dem Element gehörige Attribute
- Attribute können über ihre Position (Index) oder ihren Namen zugegriffen werden
- endElement ähnlich, jedoch ohne attributes

characters

```
public void characters(char[] buffer,
                       int offset,
                       int length)
            throws SAXException
```

buffer: Liste von Zeichen

String s = new String(buffer, offset, length);

Beispiel

```
<priceList>
   <coffee>
         <name>
                  Mocha Java
         </name>
         <price>
                  11.95
         </price>
   </coffee>
</priceList>
```

- Aufgabe: Gib den Preis von Mocha Java aus!
- Hierfür benötigen wir zwei Dinge:
  1. einen SAX-Parser
  2. passende Callback-Methoden

Wie bekomme ich einen SAX-Parser?


```
SAXParserFactory factory = SAXParserFactory.newInstance();
```

- liefert eine SAXParserFactory

```
SAXParser saxParser = factory.newSAXParser();
```

- liefert einen SAXParser

```
saxParser.parse("priceList.xml", handler);
```

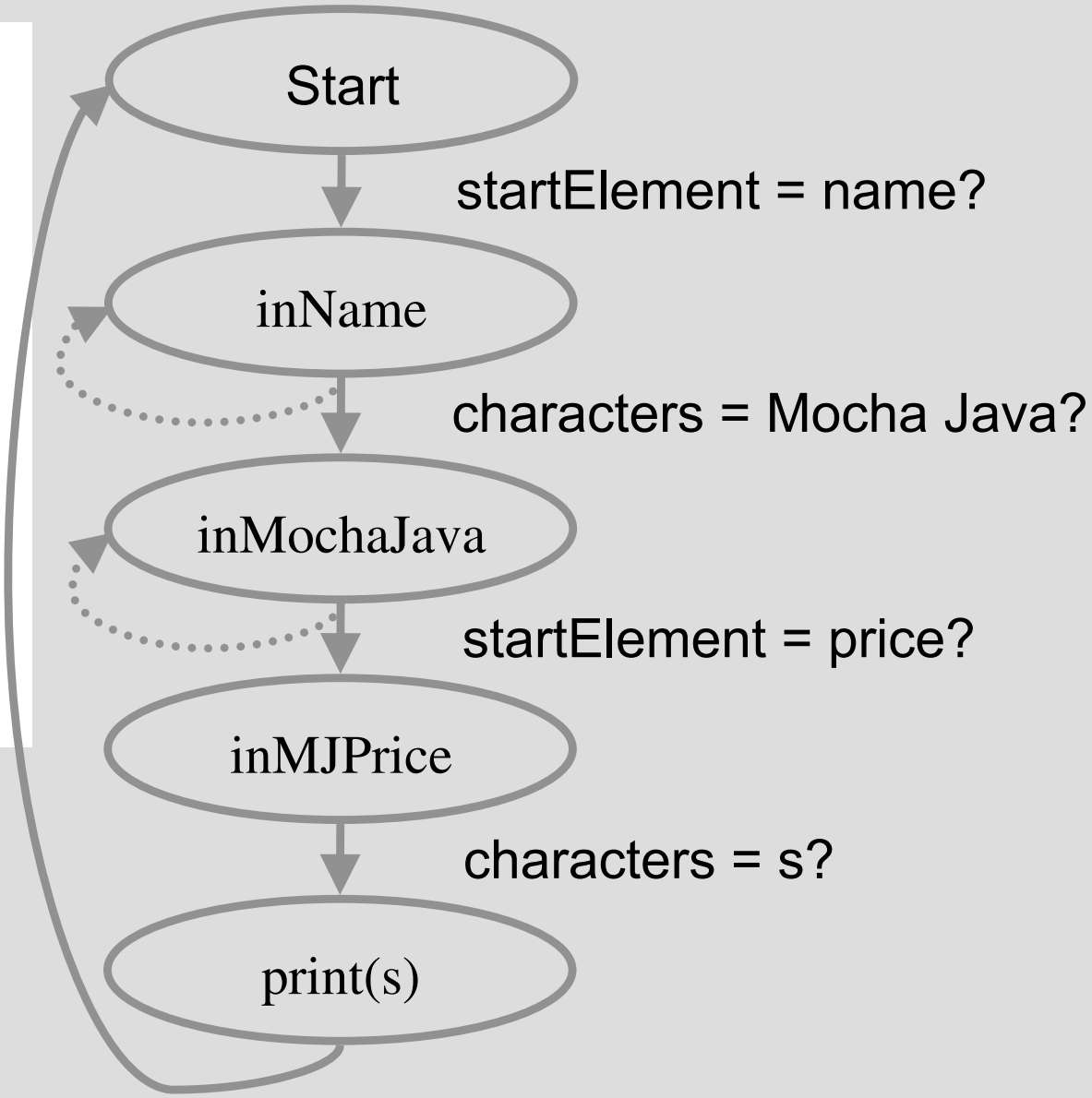
- stößt SAX-Parser an
- priceList.xml: zu parsende Datei, kann auch URL oder Stream sein
- handler: Instanz von DefaultHandler, implementiert Callback-Funktionen

Exkurs: Factory Method

- Entwurfsmuster aus „Design Patterns“ von Gamma, Helm, Johnson, Vlissides (1995)
- liefert ein Objekt
- Objekt ist Instanz einer abstrakten Klasse oder einem Interface.
- abstrakte Klasse / Interface von mehreren Klassen implementiert
- Beispiel: `Iterator i = list.iterator();`
- Beispiel: `SAXParser saxParser = factory.newSAXParser();`

Funktionsweise der Callback-Methoden

```
<priceList>
   <coffee>
         <name>
                  Mocha Java
         </name>
         <price>
                  11.95
         </price>
   </coffee>
</priceList>
```



- Zustände als boolesche Variablen

Die Callback-Methoden in Java


```
public void startElement(..., String elementName, ...) {
if (elementName.equals("name")){     inName = true;    }
else if (elementName.equals("price") && inMochaJava ){
   inMJPrice = true;
   inMochaJava = false;    } }
```

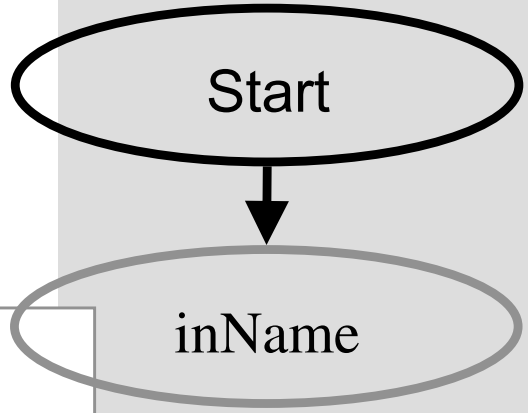
```
public void characters(char [] buf, int offset, int len) {
String s = new String(buf, offset, len);
if (inName && s.equals("Mocha Java")) {
   inMochaJava = true;
   inName = false;    }
else if (inMJPrice) {
   System.out.println("The price of Mocha Java is: " + s);
   inMJPrice = false;      }    }
```

Start: Auf <name> warten

```
public void startElement(..., String elementName, ...){
if (elementName.equals("name")){    inName = true;   }
else if (elementName.equals("price") && inMochaJava ){
   inMJPrice = true;
   inMochaJava = false;   } }
```

```
<name>Mocha Java</name>
<price>11.95</price>
```

```
public void characters(char [] buf, int offset, int len) {
String s = new String(buf, offset, len);
if (inName && s.equals("Mocha Java")) {
   inMochaJava = true;
   inName = false
else if (inMJPric
   System.out.pri
   inMJPrice = fa
```



Start

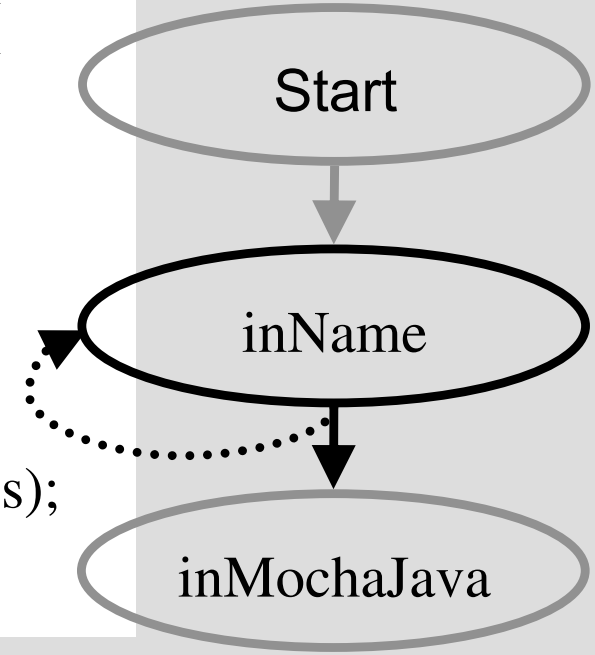
- Anfangszustand
- keine eigene Zustandsvariable
- alle Zustandsvariablen = false

inName: Auf "Mocha Java" warten

```
public void startElement(..., String elementName, ...){
if (elementName.equals("name")){     inName = true;    }
else if (elementName.equals("price") && inMochaJava ){
    inMJPrice = true;
    inMochaJava = false;    } }
```

```
<name>Mocha Java</name>
<price>11.95</price>
```

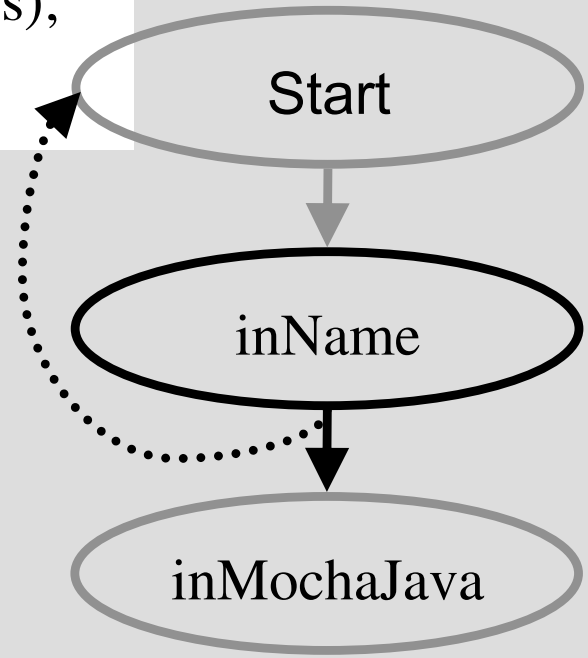
```
public void characters(char [] buf, int offset, int len) {
String s = new String(buf, offset, len);
if (inName && s.equals("Mocha Java")) {
    inMochaJava = true;
    inName = false;    }
else if (inMJPrice) {
    System.out.println("The price of Mocha Java is: " + s);
    inMJPrice = false;      }    }
```



Eine bessere Alternative

```
public void characters(char [] buf, int offset, int len) {
String s = new String(buf, offset, len);
if (inName) { if (s.equals("Mocha Java")) {
                    inMochaJava = true;
                    inName = false;  }
                else inName = false; }
else if (inMJPrice) {
   System.out.println("The price of Mocha Java is: " + s);
   inMJPrice = false;     }    }
```

```
<name>Mocha Java</name>
<price>11.95</price>
```

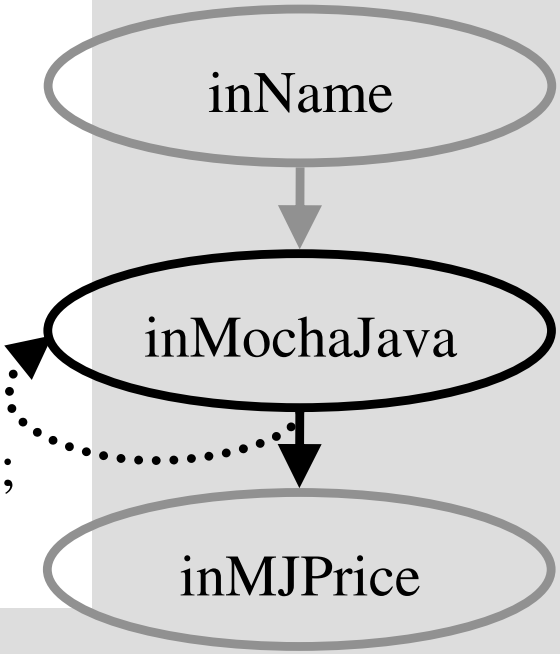


inMochaJava: Auf <price> warten

```
public void startElement(..., String elementName, ...){
if (elementName.equals("name")){     inName = true;    }
else if (elementName.equals("price") && inMochaJava ){
    inMJPrice = true;
    inMochaJava = false;    } }
```

```
<name>Mocha Java</name>
<price>11.95</price>
```

```
public void characters(char [] buf, int offset, int len) {
String s = new String(buf, offset, len);
if (inName && s.equals("Mocha Java")) {
    inMochaJava = true;
    inName = false;    }
else if (inMJPrice) {
    System.out.println("The price of Mocha Java is: " + s);
    inMJPrice = false;      }    }
```

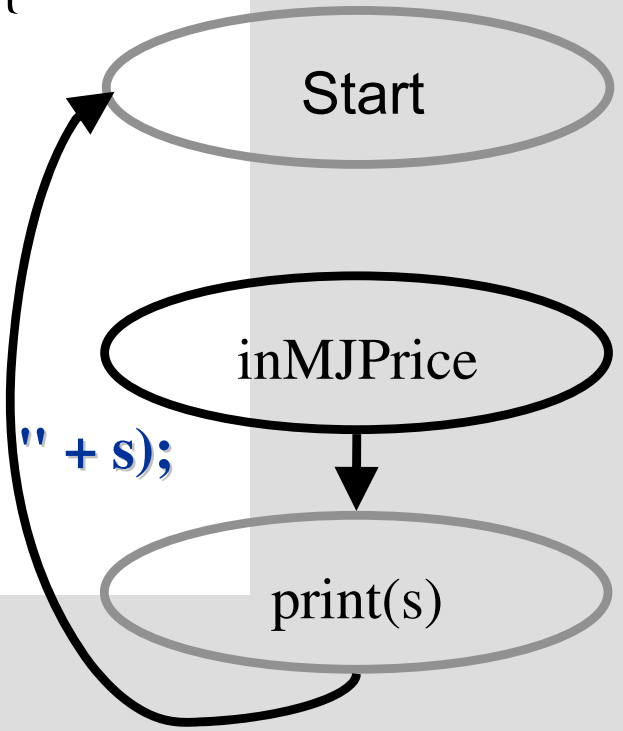


inMJPrice: Preis ausgeben

```
public void startElement(..., String elementName, ...){
if (elementName.equals("name")){     inName = true;    }
else if (elementName.equals("price") && inMochaJava ){
   inMJPrice = true;
   inMochaJava = false;    } }
```

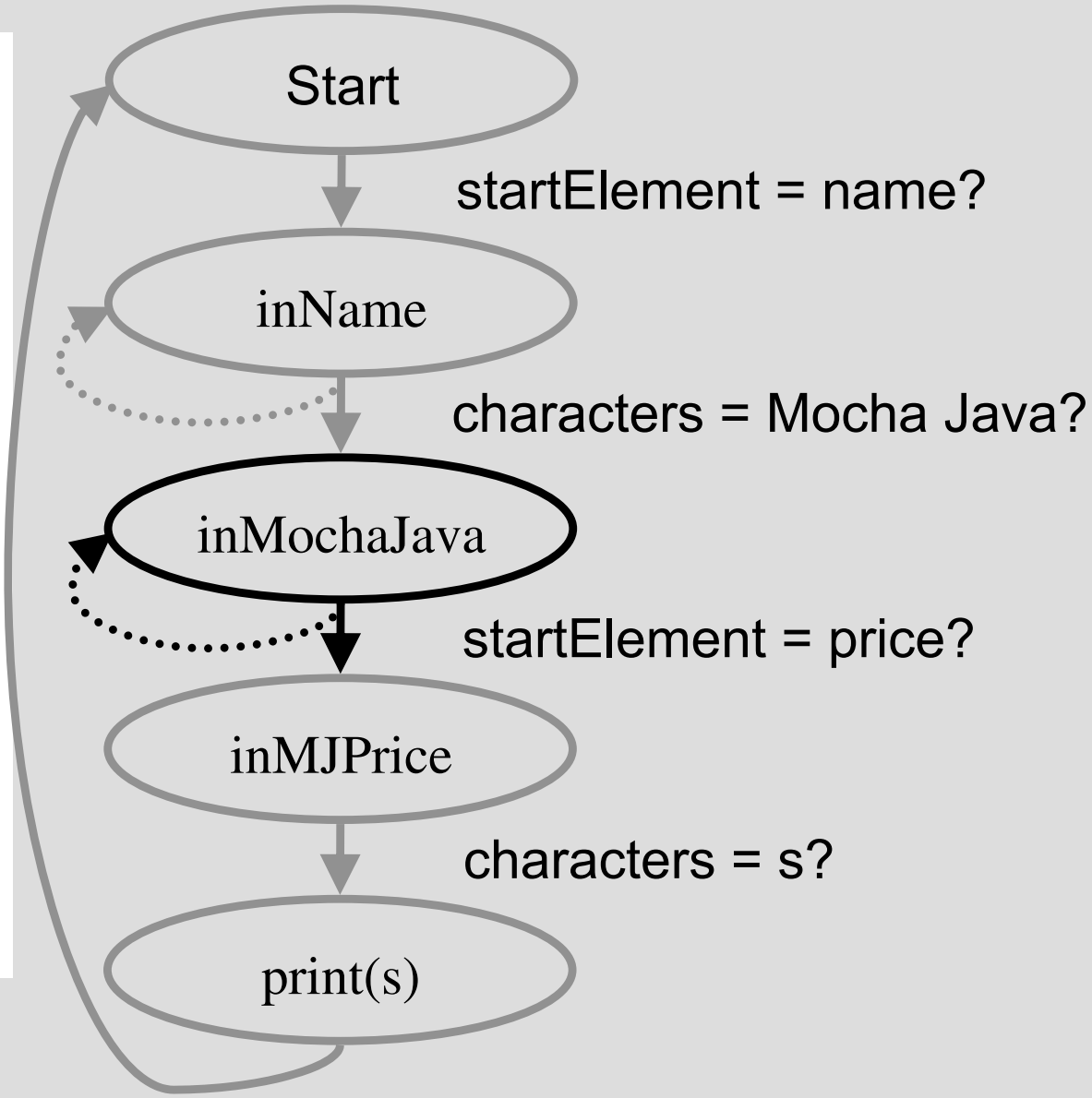
```
<name>Mocha Java</name>
<price>11.95</price>
```

```
public void characters(char [] buf, int offset, int len) {
String s = new String(buf, offset, len);
if (inName && s.equals("Mocha Java")) {
   inMochaJava = true;
   inName = false;    }
else if (inMJPrice) {
   System.out.println("The price of Mocha Java is: " + s);
   inMJPrice = false;     }   }
```



Fehlerbehandlung

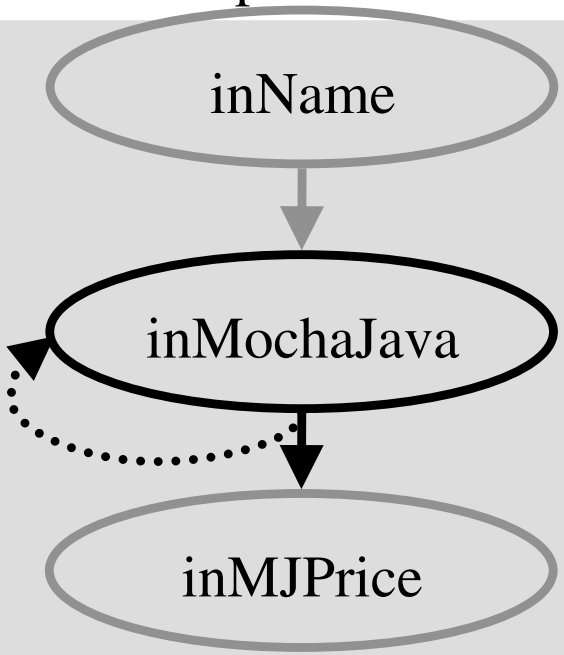
```
<priceList>
   <coffee>
         <name>
               Mocha Java
         </name>
         <name>
               MS Java
         </name>
         <price>
               11.95
         </price>
   </coffee>
</priceList>
```



Fehlerbehandlung

```
public void startElement(..., String elementName, ...){
if (elementName.equals("name")){     inName = true;    }
else if (elementName.equals("price") && inMochaJava ){
   inMJPrice = true;
   inMochaJava = false;   } }
```

```
<name>Mocha Java</name>
<name>MS Java</name>
<price>11.95</price>
```



- inMochaJava erwartet <price>
- kommt stattdessen <name>, wird aktueller Zustand inMochaJava *nicht* verändert
- kommt danach <price>, wird aktueller Zustand inMJPrice

⇨ Preis von MS Java wird ausgegeben!

Fehlerbehandlung

- SAX-Parser überprüft immer Wohlgeformtheit eines XML-Dokumentes.
- kann aber auch die *Zulässigkeit* bzgl. einer DTD oder eines Schema überprüfen
- Schema kann z.B. $(\text{name price})^+$ verlangen

⇨ Syntax- und Strukturfehler kann bereits der SAX-Parser abfangen

⇨ Callback-Methoden können dann von wohlgeformten und zulässigen Dokument ausgehen.

Vor- und Nachteile von SAX

+ sehr effizient, auch bei großen XML-Dokumenten
– Kontext (Parse-Baum) muss von Anwendung selbst verwaltet werden.
– abstrahiert nicht von XML-Syntax
– nur Parsen möglich, keine Modifikation oder Erstellung von XML-Dokumenten

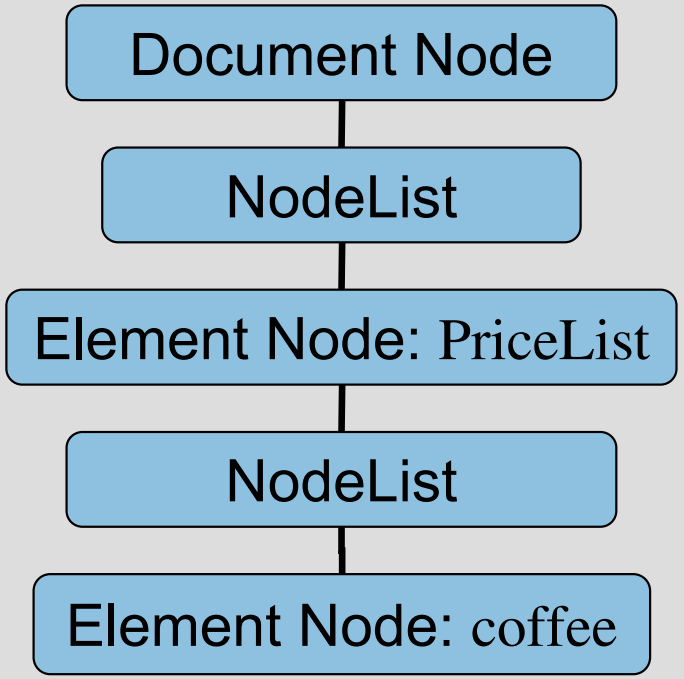
DOM-Parser

Document Object Model (DOM)

- genau genommen *kein* Parser, sondern abstrakte Schnittstelle zum Zugreifen, Modifizieren und Erstellen von Parse-Bäumen
- W3C-Standard
- unabhängig von Programmiersprachen
- nicht nur für XML-, sondern auch für HTML-Dokumente
- im Ergebnis aber Einschritt-Pull-Parser

DOM-Parse-Bäume

```
<?xml version="1.0" ?>
<priceList>
  <coffee>
    <name>Mocha Java</name>
    <price>11.95</price>
  </coffee>
</priceList>
```

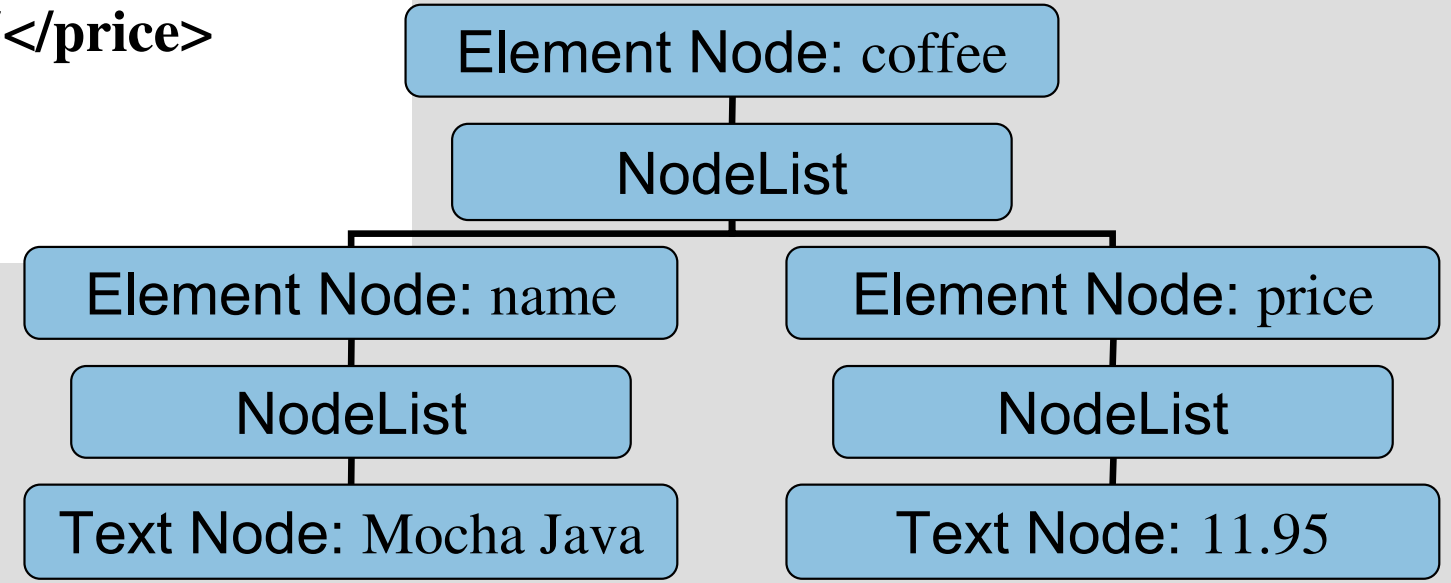


- Beachte: Dokument-Wurzel (Document Node) ≠ priceList
- Document Node: virtuelle Dokument-Wurzel, um z.B. version="1.0" zu repräsentieren
- Document Node und Element Node immer NodeList als Kind

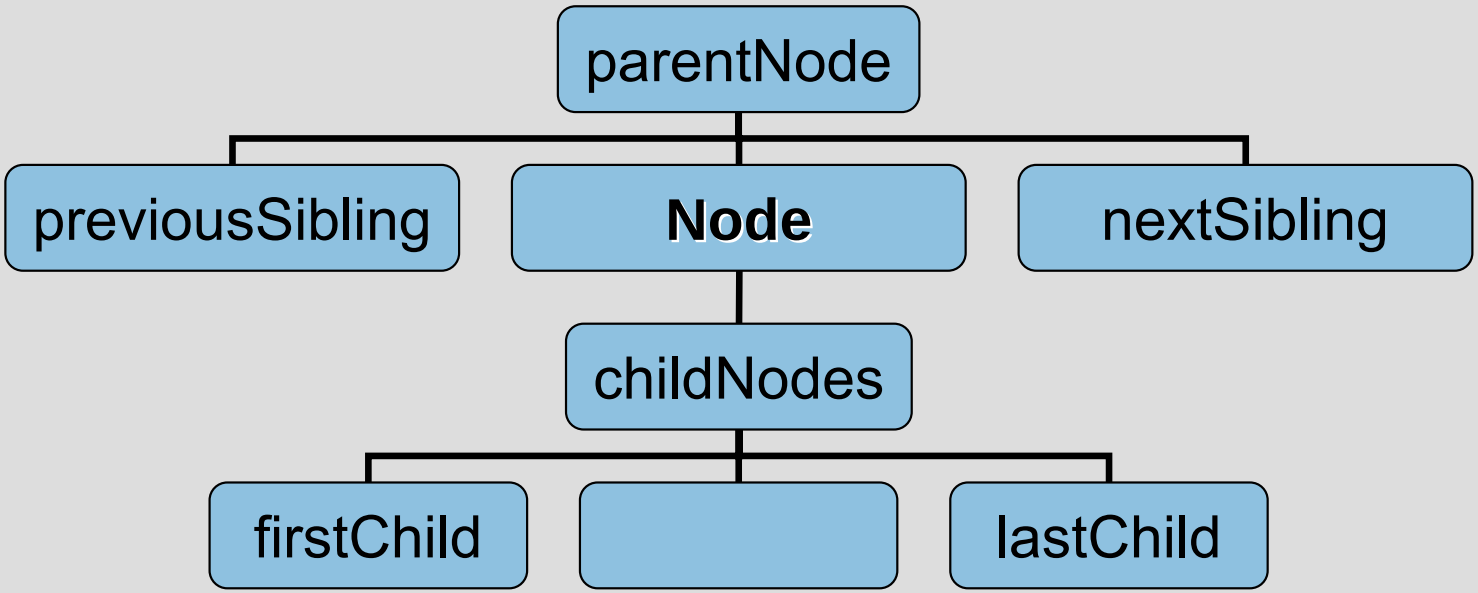
Rest des Parse-Baumes

```
<?xml version="1.0" ?>
<priceList>
  <coffee>
    <name>Mocha Java</name>
    <price>11.95</price>
  </coffee>
</priceList>
```

- Beachte: PCDATA wird als eigener Knoten dargestellt.



Navigationsmodell



- Direkter Zugriff über Namen auch möglich:
  getElementsByTagName

Beispiel

```
<priceList>
   <coffee>
         <name>
                  Mocha Java
         </name>
         <price>
                  11.95
         </price>
   </coffee>
</priceList>
```

- Aufgabe: Gib des Preis von Mocha Java aus!
- Hierfür benötigen wir zwei Dinge:
  1. einen DOM-Parser
  2. eine passende Zugriffsmethode

Wie bekomme ich einen DOM-Parser?


```
DocumentBuilderFactory factory =  DocumentBuilderFactory.newInstance();
```

- liefert DocumentBuilderFactory

```
DocumentBuilder builder = factory.newDocumentBuilder();
```

- liefert DOM-Parser

```
Document document = builder.parse("priceList.xml");
```

- DOM-Parser hat Methode parse().
- liefert in einem Schritt kompletten DOM-Parse-Baum

Wie sehen die Zugriffsmethoden aus?


```
NodeList coffeeNodes = document.getElementsByTagName("coffee");
for (int i=0; i < coffeeNodes.getLength(); i++) {
 thisCoffeeNode = coffeeNodes.item(i);
 Node thisNameNode = thisCoffeeNode.getFirstChild();
 String data = thisNameNode.getFirstChild().getNodeValue();
 if (data.equals("Mocha Java")) {
   Node thisPriceNode = thisNameNode.getNextSibling();
   String price =  thisPriceNode.getFirstChild().getNodeValue();
   break; } }
```

= Java-Programm, das DOM-Methoden benutzt

Gib mir die Liste aller coffee-Elemente!


```
NodeList coffeeNodes = document.getElementsByTagName("coffee");
```

- getElementsByTagName: direkter Zugriff auf Elemente über ihren Namen
- egal, wo Elemente stehen
- Resultat immer eine NodeList

```
<?xml version="1.0" ?>
<priceList>
  <coffee>
    <name>Mocha Java</name>
    <price>11.95</price>
  </coffee>
</priceList>
```

Betrachte alle Elemente der coffee-Liste!


```
NodeList coffeeNodes = document.getElementsByTagName("coffee");
for (int i=0; i < coffeeNodes.getLength(); i++) {
 thisCoffeeNode = coffeeNodes.item(i);
 …
 }
```

```
<?xml version="1.0" ?>
<priceList>
 <coffee>
    <name>Mocha Java</name>
    <price>11.95</price>
 </coffee>
</priceList>
```

coffeeNodes.item(0)

Gib mir erstes Kind-Element von coffee!

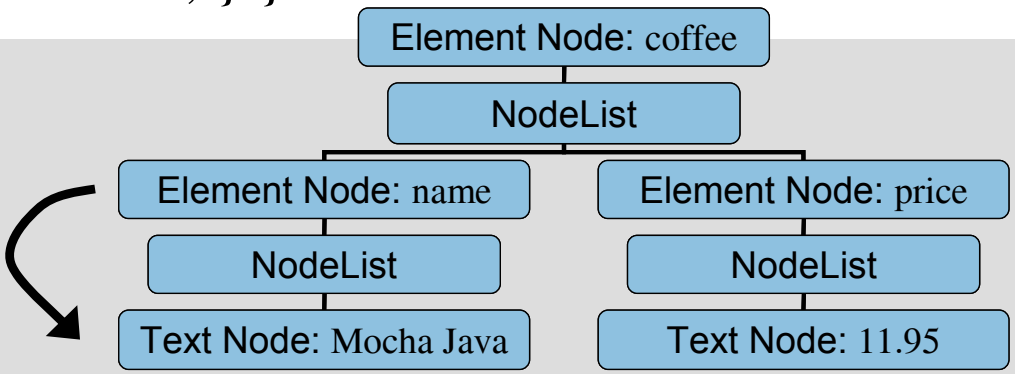

```
NodeList coffeeNodes = document.getElementsByTagName("coffee");
for (int i=0; i < coffeeNodes.getLength(); i++) {
 thisCoffeeNode = coffeeNodes.item(i);
 Node thisNameNode = thisCoffeeNode.getFirstChild();
 String data = thisNameNode.getFirstChild().getNodeValue();
 if (data.equals("Mocha Java")) {
   Node thisPriceNode = thisCoffeeNode.getNextSibling();
   String price =  thisPriceNode.getFirstChild(
   break; } }
```

firstChild

```
<?xml version="1.0" ?>
<priceList>
 <coffee>
  <name>Mocha Java</name>
   <price>11.95</price>
 </coffee>
</priceList>
```

Gib mir den Inhalt von name!

```
NodeList coffeeNodes = document.getElementsByTagName("coffee");
for (int i=0; i < coffeeNodes.getLength(); i++) {
 thisCoffeeNode = coffeeNodes.item(i);
 Node thisNameNode = thisCoffeeNode.getFirstChild();
 String data = thisNameNode.getFirstChild().getNodeValue();
 if (data.equals("Mocha Java")) {
   Node thisPriceNode = thisCoffeeNode.getNextSibling();
   String price =  thisPriceNode.getFirstChild(
   break; } }
```




```
<?xml version="1.0" ?>
<priceList>
 <coffee>
    <name>Mocha Java</name>
    <price>11.95</price>
 </coffee>
</priceList>
```

firstChild

Gib mir das Geschwister-Element!

```
NodeList coffeeNodes = document.getElementsByTagName("coffee");
for (int i=0; i < coffeeNodes.getLength(); i++) {
 thisCoffeeNode = coffeeNodes.item(i);
 Node thisNameNode = thisCoffeeNode.getFirstChild();
 String data = thisNameNode.getFirstChild().getNodeValue();
 if (data.equals("Mocha Java")) {
   Node thisPriceNode = thisNameNode.getNextSibling();
   String price =  thisPriceNode.getFirstChild(
   break; } }
```

nextSibling

```
<?xml version="1.0" ?>
<priceList>
  <coffee>
    <name>Mocha Java</name>
    <price>11.95</price>
  </coffee>
</priceList>
```

Gib mir den Inhalt von price!

```
NodeList coffeeNodes = document.getEle
for (int i=0; i < coffeeNodes.getLength(); i
 thisCoffeeNode = coffeeNodes.item(i);
 Node thisNameNode = thisCoffeeNode.ge
 String data = thisNameNode.getFirstChild
 if (data.equals("Mocha Java")) {
   Node thisPriceNode = thisNameNode.getNextSibling();
   String price =  thisPriceNode.getFirstChild().getNodeValue();
   break; } }
```

```
<?xml version="1.0" ?>
<priceList>
  <coffee>
    <name>Mocha Java</name>
    <price>11.95</price>
  </coffee>
</priceList>
```

Vor- und Nachteile von DOM

+ Kontext (Parse-Baum) muss *nicht* von Anwendung verwaltet werden.
+ direkter Zugriff auf Elemente über ihre Namen
+ nicht nur Parsen, sondern auch Modifikation und Erstellung von XML-Dokumenten
– speicherintensiv
– abstrahiert nicht von XML-Syntax

SAX oder DOM?

SAX

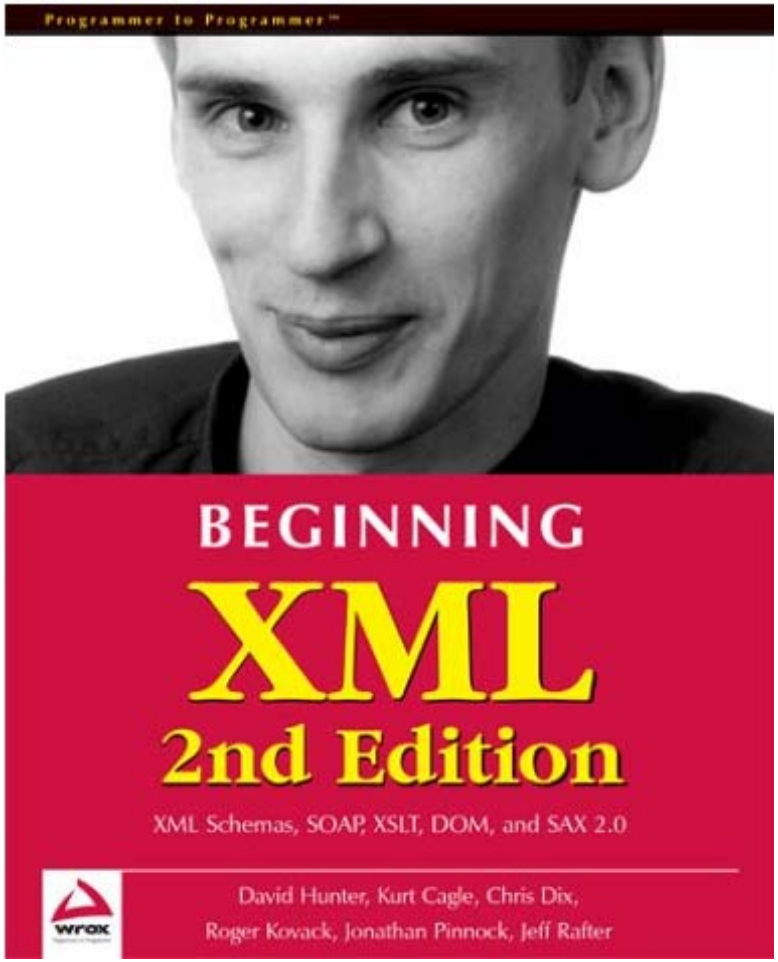
- geeignet, um gezielt bestimmte Teile von XML-Dokumenten herauszufiltern, ohne zu einem späteren Zeitpunkt andere Teile des Dokumentes zu benötigen
- nur Parsen, kein Erstellen oder Modifizieren von XML-Dokumenten

DOM

- geeignet, um auf unterschiedliche Teile eines XML-Dokumentes zu verschiedenen Zeitpunkten zuzugreifen
- auch Erstellen und Modifizieren von XML-Dokumenten

Literatur

- Hunter, David; Cagle, Kurt; Dix, Chris: Beginning XML XML Schemas, SOAP, XSLT, DOM, and SAX 2.0 2nd ed. 2001. Wrox Press 1-86100-559-8



- Empfehlenswertes Skript einer anderen XML-Vorlesung: http://www.jeckle.de/vorlesung/xml/
- mehrere Interaktive XML-Kurse http://www.zvon.org